Michael Weber

Allgemeine Konzepte zur software-technischen Unterstützung verschiedener Petrinetz-Typen

Dissertation

Allgemeine Konzepte zur software-technischen Unterstützung verschiedener Petrinetz-Typen

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(*doctor rerum naturalium*, Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von

Herrn Dipl.-Inf.
Michael Weber

geboren am 15. Mai 1968 in Mühlhausen/Thür.

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Jürgen Mlynek

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Elmar Kulke

1. Gutachter	Prof. Dr. Wolfgang Reisig	Humboldt-Universität zu Berlin, Institut für Informatik
2. Gutachter	Prof. Dr. Eike Best	Universität Oldenburg, Fachbereich Informatik
3. Gutachter	Prof. Dr. Holger Schlingloff	Humboldt-Universität zu Berlin, Institut für Informatik

eingereicht am	4. Juni 2002
Tag der mündlichen Prüfung	16. Dezember 2002

Diese Arbeit wurde mit dem Textsatzsystem TEX (`pdftex` mit optischem Randausgleich), der Dokumentenbeschreibungssprache LATEX und darauf aufbauenden Makropaketen (insbesondere KOMA-Script) sowie aus der European Computer Modern 11/13,6 Pkt. gesetzt.

Kurzfassung

Petrinetze werden in vielen Bereichen als Modellierungstechnik verwendet. Die verschiedenen Einsatzgebiete und Modellierungsziele erfordern dabei unterschiedliche Typen von Petrinetzen. Einen Petrinetz-Typ kennzeichnen – neben den üblichen Stellen, Transitionen und Kanten – eine Menge zusätzlicher, spezifischer Elemente, sowie eine spezifische Schaltregel.

In der Literatur findet man zahlreiche verschiedene Petrinetz-Typen. Diese Vielfalt an Petrinetz-Typen lässt sich nicht ohne weiteres überblicken. Deshalb fehlt es auch nicht an Versuchen, allgemeine Petrinetz-Typen oder Klassifikationen – auch einzelner Aspekte – zu etablieren. Allerdings erfassen die bisherigen Ansätze nur einen kleinen Teil aller Petrinetz-Typen. Unser semantisch orientierter Klassifizierungsansatz des Petrinetz-Hyperwürfels umfasst deutlich mehr Petrinetz-Typen und erhebt den Anspruch, universell zu sein.

Der Petrinetz-Hyperwürfel hat einen syntaktisch orientierten Klassifizierungsansatz als Grundlage. Dieser Ansatz führt einerseits zum Vorschlag der *Petri Net Markup Language*. Damit können Petrinetze aller Typen einheitlich beschrieben werden.

Andererseits führt derselbe Ansatz zu einer Basis für Petrinetz-Werkzeuge, in der die einzelnen Teile eines Petrinetz-Typs unabhängig voneinander implementiert werden. Der Petrinetz-Kern ist eine derartige Basis mit dessen Hilfe Petrinetz-Werkzeuge gebaut werden. Er implementiert Konzepte, die allen Petrinetzen gemein sind, unabhängig von konkreten Petrinetz-Typen. Gemeinsam mit dem Petrinetz-Hyperwürfel bildet der Petrinetz-Kern ein weiteres Basiswerkzeug für einen parametrisierten Petrinetz-Typ mit einer parametrisierten Schaltregel.

Die *Petri Net Markup Language* und der Petrinetz-Kern sind die wesentlichen Beiträge der vorliegenden Arbeit. Gemeinsam bilden sie ein mächtiges Grundgerüst für Petrinetz-Werkzeuge beliebiger Petrinetz-Typen.

Schlagwörter:

Petrinetze, Petrinetz-Typen, Petrinetz-Kern, Petri Net Markup Language, Petrinetz-Würfel

Abstract

Petri nets are widely used for modelling systems. The different areas and goals require different types of Petri nets. Each Petri net contains beside places, transitions, and arcs several further specific elements. Furthermore, a Petri net type defines a specific firing rule.

There are many different Petri net types. It is not easy to have a general view on this bulk of Petri net types. Thus, there are attempts to establish general Petri net types or classifications of Petri net types (even of particular aspects). But, current approaches include only a few of all Petri net types. Our approach is a classification by semantics of Petri nets. We call this classification Petri Net Hypercube. It is meant to be universal for all Petri net types.

A syntactical classification approach is the base of the Petri Net Hypercube. This approach leads on the one hand to the proposal of the Petri Net Markup Language. This language describes Petri nets of all types.

On the other hand, the same approach leads to a base of Petri net tools. The parts of a Petri net type are implemented in this base independently of each other. The Petri Net Kernel is such a base for building Petri net tools. It implements those concepts which are general concepts of each Petri net. The Petri Net Kernel forms together with the Petri Net Hypercube a further basic Petri net tool for a parameterized Petri net type with a parameterized firing rule.

The Petri Net Markup Language and the Petri Net Kernel are the main contributions of this thesis. Together, they are a powerful base for Petri net tools of each Petri net type.

Keywords:

Petri nets, Petri net types, Petri Net Kernel, Petri Net Markup Language, Petri Net Cube

Vorwort

Die vorliegende Arbeit sowie insbesondere der Petrinetz-Kern (PNK) und die *Petri Net Markup Language* (PNML) entstanden im Rahmen der von der Deutschen Forschungsgemeinschaft (DFG) geförderten Forschergruppe „Petrinetz-Technologie“. Ihr gehörte ich von Februar 1997 bis März 2002 als wissenschaftlicher Mitarbeiter an. Die Forschergruppe beschäftigte sich mit dem Einsatz von Petrinetzen als Technologie zum Bau von Software-Systemen.

Eine umfangreiche Arbeit wie die vorliegende Dissertation kann nicht unbeeinflusst von vielerlei Eindrücken entstehen. Neben den in dieser Arbeit explizit und implizit dokumentierten, sind es die nicht sichtbaren Einflüsse, die mich vor allem persönlich geprägt haben und so nicht unerheblich zum Ergebnis beigetragen haben. Im folgenden seien die Urheber der wesentlichen Einflüsse gewürdigt.

Meinem Doktorvater Prof. Wolfgang Reisig danke ich sehr für den gewohnt ungeduldig geduldigen Ansporn, meine Gedanken und Ergebnisse verständlich zu präsentieren. Ihm verdanke ich ein ideales Arbeitsklima zwischen Anspruch und Gelassenheit. Prof. Eike Best danke ich für die Übernahme des zweiten Gutachtens und seine Forderung nach dem „Roten Faden“. Prof. Holger Schlingloff danke ich für das dritte Gutachten; ihm verdanke ich zukunftsweisende Fragen, die zweifellos den weiteren Fortgang der Projekte PNK und PNML beeinflussen werden.

Eine besondere Unterstützung fand ich in Ekkart Kindler. Ihm danke ich für gemeinsame Arbeiten, beständige Ermutigungen und Anregungen, geistige Diskussionen sowie Hinweise zur formalen Darstellung in meiner Arbeit. Wichtige Impulse zur Entwicklung des PNK gingen von ihm aus. Dem PNK-Team danke ich für seine bis in die nahe Zukunft fortgesetzte Arbeit am PNK. Seinen Mitgliedern ist der erreichte Zustand – insbesondere die Implementation – hoch anzurechnen. Dem Team gehör(t)en neben Ekkart Kindler und Bodo Hohberg an: Frank Oschmann (vielen Dank für kompetente Implementierungsarbeit), Alexander Grünewald (vielen Dank für die Beiträge zur Reimplementierung und Ideen zur graphischen Nutzerschnittstelle), Erik Fischer (vielen Dank für die Implementierungsarbeiten zum Petrinetz-Hyperwürfel), Matthias Jüngel (vielen Dank für die entscheidende Anregung zur PNML), Ines Schwenzer, Jens Hauptmann, Abdourahaman, Reiner Schulz und Raimund Klein. Alexander Grünewald danke ich außerdem für die gründliche Durchsicht der Abbildungen und Programme in der vorliegenden Arbeit. Er fand einige Formulierungsfehler und machte mich auf Unzulänglichkeiten im formalen Teil aufmerksam.

Den Mitgliedern der Kaffeerunde am Lehrstuhl Theorie der Programmierung danke ich für ihre Geduld, meine halb fertigen Gedanken ertragen zu haben. Prof. Bodo Hohberg danke ich für einen wichtigen Hinweis zur Belegung in der parametrisierten Schaltregel. Stundenlange Telefongespräche mit Hagen Völzer brachten mehr Klarheit in die formale Darstellung. Stephan Roch und Karsten Schmidt verdanke ich eine Einführung in Signal-Netzsysteme sowie die kritische Begleitung des PNK und der PNML. Axel Martens hat gezeigt, dass der PNK für sinnvolle Projekte außerhalb der Kompetenzen des PNK-Teams verwendet werden kann. Vorgenannte Mitglieder der Kaffeerunde sowie Ekkart Kindler, Birgit Heene, Adrianna Alexander und Sibylle Peuker haben immer geglaubt (manchmal im Gegensatz zu mir), dass ich es schaffen werde, und mich immer wieder angespornt.

Der Deutschen Forschungsgemeinschaft (DFG) danke ich für die großzügige finanzielle Unterstützung des Projektes „Petrinetz-Technologie“ ohne die die Entwicklung des PNK und der PNML nicht möglich gewesen wäre.

Ich danke Marko Mäkelä für Hinweise auf Publikationen des Projektes Maria und Fehler im Textsatz. Frank Schramm hat dankenswerterweise eine frühe Vorversion der Arbeit durchgesehen und mich auf die Standardisierungsbemühungen für ein kompaktes XML-Format hingewiesen.

Nadja Richter, Katrin Dahme sowie Hella und Volker Weber haben den Text auf Rechtschreibung und Zeichensetzung untersucht. Ihnen danke ich sehr, dass sie sich durch den gesamten Text „gekämpft“ haben. Meinen Freunden und Verwandten danke ich für Zerstreuung und spitzfindige Fragen.

Und schließlich danke ich Nadja Richter dafür, dass sie meine Arbeit von Anfang an wohlwollend und mit aller Kraft unterstützt hat. Sie war mir die wichtigste Stütze insbesondere in den „heißen“ und heiklen Phasen der Arbeit. Unsere gemeinsamen Söhne und sie haben mir immer wieder verständnisvoll gezeigt, was wirklich wichtig ist im Leben. Das gab mir die nötige Gelassenheit, alle Widrigkeiten zu meistern.

Berlin, im Januar 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellen- und Programmverzeichnis

Tabellenverzeichnis

Programmverzeichnis

1 Einleitung

Man versteht etwas nicht wirklich, wenn man nicht versucht, es zu implementieren. DONALD E. KNUTH

In der vorliegenden Arbeit geht es um die software-technische Unterstützung der zahlreichen verschiedenen Petrinetz-Typen. Es werden Konzepte vorgestellt, die alle Petrinetz-Typen einheitlich behandeln und ihre Unterschiede herausarbeiten. Damit profitiert der Software-Entwickler eines Petrinetz-Werkzeuges von der Entwicklung anderer Werkzeuge für „ähnliche“ Petrinetz-Typen. Die beiden wesentlichen Ergebnisse der Arbeit sind die *Petri Net Markup Language* (PNML) und der Petrinetz-Kern (PNK). PNML ist eine Beschreibungssprache für beliebige Petrinetze. Der PNK ist eine Infrastruktur zum Bau von Petrinetz-Werkzeugen. Gemeinsam bilden sie ein mächtiges Grundgerüst für Petrinetz-Werkzeuge beliebiger Petrinetz-Typen.

Petrinetze

Petrinetze werden heute in vielen Bereichen zur Systemmodellierung eingesetzt. Geschäftsprozesse lassen sich beispielsweise sehr gut mit Petrinetzen modellieren. Durch die Modellierung lokaler Systemzustände und -übergänge sind Petrinetze besonders für die Modellierung und Analyse verteilter Systeme geeignet. Dies wurde in der Vergangenheit mehrfach nachgewiesen. Ein häufig genannter Vorteil von Petrinetzen ist ihre grafische Notation. Sie besteht aus den vier Grundelementen *Stelle*, *Transition*, *Kante* und *Marke*. Stellen, Transitionen und Kanten modellieren statische Verhältnisse eines Systems. Stellen sind passive Elemente und stellen Zustände des Systems dar. Transitionen sind aktive Elemente und stellen Transformationen von Zuständen dar. Kanten sind Relationen. Eine Kante stellt die Beziehung zwischen einer Stelle und einer Transition her. Petrinetze werden als Graphen mit Stellen und Transitionen als Knoten sowie gerichteten Kanten dargestellt. Eine Kante verbindet jeweils eine Stelle mit einer Transition bzw. umgekehrt.

Das vierte Grundelement in Petrinetzen sind Marken, die die dynamischen Verhältnisse eines Systems modellieren. Eine Marke (auch Token genannt) befindet sich auf einer Stelle. Sie modelliert beispielsweise eine verfügbare Ressource oder ein Objekt des Systems, die bzw. das sich in dem von der Stelle modellierten Zustand befindet. Marken bilden gemeinsam mit den Stellen die tatsächlichen Zustände des gesamten Systems ab. Eine Transition konsumiert bzw. produziert Marken von einer Stelle bzw.

auf einer Stelle entsprechend der Kante zu der Stelle. Wir sprechen dann vom *Schalten* einer Transition. Eine Transition beeinflusst beim Schalten Teilzustände eines Systems, d. h. nur die Stellen, die mit der Transition über Kanten direkt verbunden sind.

Mit Petrinetzen modellierte Systeme erreichen durch ihre grafische Notation einen hohen Grad an Anschaulichkeit. Außerdem ist die Schaltregel von Petrinetzen mathematisch klar definiert. Da ein Petrinetz ein Graph ist, lassen sich graphentheoretische Verfahren zur Analyse von Petrinetzen einsetzen.

Petrinetze sind weniger dafür geeignet, modellierte Systeme direkt zu implementieren, da das Petrinetzen zu Grunde liegende Architekturmodell nur selten mit der Architektur der Zielsysteme übereinstimmt. Petrinetze haben vor allem dann Erfolg, wenn sie in Programme des Zielsystems übersetzt werden [CPN00a] oder wenn sie direkt interpretiert werden können. Workflowsysteme werden häufig mit Petrinetzen modelliert [ADO00]. Diese werden dann in speziellen Interpretern (z. B. COSA [COS01]) ausgeführt. Petrinetze werden auch zur Analyse für verteilte bzw. parallele Programme [Gra97] eingesetzt.

Es gibt viele verschiedene *Petrinetz-Typen*, zum Beispiel Stellen-Transitions-Netze [Rei87], Bedingungs-Ereignis-Netze [GLT80], Prädikat-Transitions-Netze [GLT80], *Coloured Petri Nets* [Jen92], Algebraische Petrinetze [Rei91a], Signal-Netzsysteme [HL00], Zeit-Petrinetze [Sta95] u. v. m. Eine *Instanz* eines Petrinetz-Typs – also ein konkretes Petrinetz diesen Typs – unterscheidet sich von Netzen anderer Typen bspw. in der Art der Marken, der Schaltregel, der Verwendung anderer Beschriftungen etc. Dennoch gibt es Gemeinsamkeiten, die es rechtfertigen, alle derartigen Netze als Petrinetze zu bezeichnen. Gemeinsam sind allen Petrinetzen unterschiedlicher Typen die beiden grundsätzlich verschiedenen Arten von Knoten – Stellen und Transitionen – sowie Kanten zwischen verschiedenartigen Knoten. Bei der Definition eines Petrinetz-Typs wird immer ein Petrinetz-Graph zugrunde gelegt, der durch weitere Abbildungen und Funktionen zu einer Definition des Typs ergänzt wird. Petrinetze verschiedener Petrinetz-Typen können zuweilen die gleiche textuelle bzw. grafische Repräsentation erhalten. Wichtig ist in diesem Falle, dass sich die Betrachter auf einen Petrinetz-Typ einigen, dem das betreffende Petrinetz angehört.

Klassifikation von Petrinetz-Typen

Um verschiedene Petrinetz-Typen miteinander zu vergleichen sowie vor allem Analysemethoden und Modellierungstechniken von Petrinetzen eines Typs auf einen anderen zu übertragen, benötigt man eine Klassifikation von Petrinetz-Typen. Ein erster Ansatz sind systematische Sammlungen von Petrinetz-Typen. Der bekannteste Ansatz sind hier die Bemühungen um eine Allgemeine Petrinetz-Theorie [Pet77b, Bra80] oder verschiedene Überblicksarbeiten (z. B. in [Roz92]). Auch Arbeiten, in denen Notationen und Terminologien verschiedener Petrinetz-Typen systematisch eingeführt

werden (z. B. [BF86]), wollen wir unter diesen Ansatz einordnen. Und schließlich gibt es Sammlungen und Kataloge für Petrinetz-Werkzeuge (z. B. [Fel93, PNTo0, Stö97]), die geeignet sind, Petrinetz-Typen in Bezug auf ihren Einsatz in Werkzeugen zu vergleichen.

Ein anderer Ansatz ist, *einen* Petrinetz-Typ zu entwickeln, so dass Instanzen anderer Petrinetz-Typen auch Instanzen des *einen* sind. Ein solches allgemeines Petrinetz (z. B. [Bra92]) muss ein *high level* Netz sein und kann keine allgemeinen Aussagen (z. B. Invarianten, Erreichbarkeit) mehr treffen, die nur in speziellen Instanzen gelten.

Ein dritter Ansatz identifiziert (unabhängige) Parameter in einer Menge von Petrinetz-Typen, so dass alle Petrinetz-Typen dieser Menge durch entsprechend gewählte Parameter beschrieben werden können. Klassifikationen dieser Art wurden für spezielle Petrinetz-Typen wie Zeit-Petrinetze [Sta95] oder Kantentypen in *Coloured Petri Nets* [LC94] etabliert. Zur Klassifikation beliebiger Petrinetz-Typen wurden verschiedene parametrisierte Petrinetz-Typen vorgestellt. Abstrakte Petrinetze [Pad96], algebraisch allgemeine Petrinetze [Juh99] und der Petrinetz-Würfel [KW98] sind Beispiele für parametrisierte Petrinetz-Typen. In dieser Arbeit wird eine Klassifikation von Petrinetz-Typen mit Hilfe des Petrinetz-Hyperwürfels vorgestellt. Er baut auf dem Petrinetz-Würfel auf und erweitert ihn um weitere Dimensionen.

Eine Klassifikation von Petrinetz-Typen bestimmt Gemeinsamkeiten und Unterschiede zwischen verschiedenen Petrinetz-Typen bezüglich eines allgemeinen Rahmens vergleichbarer Begriffe. Dann ist es möglich, „Abstände“ zwischen Petrinetz-Typen bezüglich des gewählten Rahmens zu bestimmen und Ergebnisse für Netze eines Typs auf Netze anderer Typen begründet zu übertragen oder nicht zu übertragen. Klassifikationen von Petrinetz-Typen eignen sich z. B. dazu, neuartige Petrinetz-Typen zu entwickeln [Pad96] oder die Schaltregel einer Menge von Petrinetz-Typen unabhängig von konkreten Parametern zu definieren [KW98]. Aufbauend auf dem Petrinetz-Hyperwürfel wird in dieser Arbeit der implementierte Ansatz eines parametrisierten Petrinetz-Typs vorgestellt. Es wird ein parametrisierbares Petrinetz-Werkzeug diskutiert, das für einen beliebigen Petrinetz-Typ ein Petrinetz-Werkzeug mit einem grafischen Editor, ein Dateiformat zum Speichern entsprechender Netze und einen interaktiven Markenspielsimulator darstellt.

Konzepte

Zur Klassifikation von Petrinetz-Typen werden in dieser Arbeit zwei Konzepte verwendet. Das erste beruht auf der Beobachtung, dass sich die verschiedenen Petrinetz-Typen darin unterscheiden, welche Anschriften sie an Stellen, Transitionen und Kanten eines Netzes zulassen. Wir nennen solche Anschriften *Label*. Unterschiede zwischen verschiedenen Petrinetz-Typen haben ihre Ursache in der Art der Label und nicht z. B. in Eigenschaften des Graphen eines Netzes. In dieser Arbeit wird gezeigt, dass Petrinetze beliebiger Typen mit diesem Konzept beschrieben werden können. Ein

solches Konzept eignet sich gut zur Implementation eines allgemeinen Rahmens für Petrinetz-Werkzeuge für verschiedene Petrinetz-Typen.

Das zweite Konzept zur Klassifizierung von Petrinetz-Typen, das in dieser Arbeit eingeführt wird, baut auf dem Petrinetz-Würfel auf. Der Petrinetz-Würfel wird um einige Dimensionen, wie Kantentypen, eine Menge von Labeldefinitionen und eine parametrisierte Schaltregel, zu einem Petrinetz-Hyperwürfel erweitert. Damit können beliebige Petrinetz-Typen klassifiziert werden. Genauer gesagt, werden mit dem Petrinetz-Hyperwürfel die Label eines Petrinetz-Typs und ihre Beziehung zueinander beschrieben.

In dieser Arbeit wird für den Petrinetz-Hyperwürfel eine *parametrisierte Schaltregel* vorgestellt, denn es gibt eine weitere Gemeinsamkeit aller Petrinetz-Typen: ihre Netze haben Schaltverhalten. Die parametrisierte Schaltregel orientiert sich an den Ansätzen für allgemeine Petrinetz-Typen und soll zudem leicht zu implementieren sein. Sie basiert auf einer Klassifikation von Petrinetz-Schaltregeln mit Hilfe von Parametern und Eigenschaften von Labeln. Diesem Ansatz zu Folge lässt sich jede konkrete Petrinetz-Schaltregel als Instanz einer *allgemeinen* Schaltregel formulieren.

Software-technische Unterstützung

Beide Konzepte – das Labelkonzept wie auch das Klassifizierungskonzept – sind Grundlagen für weiterführende Konzepte, die ebenfalls in dieser Arbeit vorgestellt werden. Das Labelkonzept ist Grundlage der Petrinetz-Beschreibungssprache *Petri Net Markup Language* (PNML) und der Basisversion einer Infrastruktur zum Bau von Petrinetz-Werkzeugen – dem *Petrinetz-Kern* (PNK). PNML ist gleichzeitig ein Vorschlag für ein einheitliches Dateiformat für Petrinetze. Ein einheitliches Dateiformat erleichtert den Austausch von Petrinetzen zwischen verschiedenen Werkzeugen, da die Struktur einer Petrinetz-Datei und die Bedeutung einzelner Teile festgelegt ist. Mit PNML wird vorgeschlagen, die Label von Stellen, Transitionen und Kanten einheitlich zu beschreiben. Außerdem wird vorgeschlagen, die textuelle Repräsentation von Petrinetzen mit Informationen zur grafischen und strukturierten Darstellung auf einheitliche Weise zu ergänzen. PNML ist so konzipiert, dass ein Petrinetz *jeden* Typs damit beschrieben werden kann. Außerdem wird vorgeschlagen, die textuelle Repräsentation häufig benutzter Label zu standardisieren.

Für Entwickler von Petrinetz-Werkzeugen stellt sich häufig das Problem, Funktionalität implementieren zu müssen, die schon in anderen Werkzeugen (und dort zum Teil besser) implementiert wurde. In dieser Arbeit wird auch der Petrinetz-Kern (PNK) vorgestellt. Er ist eine Infrastruktur zum Bau von Petrinetz-Werkzeugen und besteht aus einer Programmierschnittstelle für Grundfunktionen eines Petrinetz-Werkzeuges, einer Anwendungssteuerung sowie einer Schnittstelle zur Definition von Petrinetz-Typen entsprechend des Labelkonzeptes. Eine weitere Programmier-

schnittstelle des PNK implementiert parametrisierte Petrinetz-Typen nach dem Konzept des Petrinetz-Hyperwürfels.

Mit beiden Schnittstellen des PNK für Petrinetz-Typen ist es einem Anwender einfach möglich, einen Petrinetz-Typ für ein Werkzeug zusammenzustellen, im besten Fall ohne selbst zu programmieren. Ein derartig erzeugtes Basiswerkzeug besteht wenigstens aus einem grafischen Editor, einem Dateiformat und einem Markenspielsimulator. Damit ist der PNK in Kombination mit dem Dateiformat PNML ein mächtiges Grundgerüst für Petrinetz-Werkzeuge.

Die Schnittstelle des PNK für parametrisierte Petrinetz-Typen validiert das Klassifizierungskonzept des Petrinetz-Hyperwürfels. Jeder Parameter steht dabei für eine der voneinander unabhängigen Dimensionen des Petrinetz-Hyperwürfels. Entsprechende konkrete Werte (implementierte Parameter) und ihre Kombination ergeben dann die formale Beschreibung eines Petrinetz-Typs und mit der Schnittstelle des PNK das Basiswerkzeug für Petrinetze diesen Typs. Aufbauend auf der parametrisierten Schaltregel des Petrinetz-Hyperwürfels lassen sich (erreichbarkeitsbasierte) Analysealgorithmen unabhängig von konkreten Petrinetz-Typen implementieren. Die Implementation für den Petrinetz-Hyperwürfel ist damit eine Basis für ein Petrinetz-Werkzeug eines *beliebigen* Petrinetz-Typs.

Gliederung

Die vorliegende Arbeit gliedert sich in zwei Teile. Der erste Teil besteht aus Kap. 2 und Kap. 3. Zunächst werden in Kap. 2 mathematische und Petrinetz-theoretische Grundlagen gelegt. Dann entwickeln wir in Kap. 3 aus der Idee des Petrinetz-Würfels den Petrinetz-Hyperwürfel. Im zweiten Teil (Kap. 4 und Kap. 5) werden die zuvor vorgestellten Konzepte mit software-technischen Methoden zu anwendbaren Petrinetz-Werkzeugen weiterentwickelt. Kapitel 4 führt die Petrinetz-Beschreibungssprache PNML und weitere netztypunabhängige Konzepte zur Behandlung großer Petrinetze und wiederverwendbarer Teile von Petrinetzen ein. Kapitel 5 stellt den Petrinetz-Kern als Infrastruktur zum Bau von Petrinetz-Werkzeugen vor. Dasselbe Kapitel wird abgeschlossen durch die Beschreibung einer Implementation für den Petrinetz-Hyperwürfel mit Hilfe des PNK.

2 Grundbegriffe

In diesem Kapitel werden einige Grundbegriffe der Mathematik und der Petrinetz-Theorie eingeführt, die später in der Arbeit benötigt werden. Wir beginnen mit mathematischen Begriffen (Abschn. 2.1). In Abschn. 2.2 folgen Grundbegriffe der Petrinetz-Theorie. Der Begriff des Petrinetz-Typs wird mit Hilfe von Beispielen erläutert. Außerdem werden dort Ergebnisse der Literatur zur Klassifikation von Petrinetz-Typen diskutiert. Schließlich wird in Abschn. 2.3 die Welt der Petrinetz-Werkzeuge betrachtet.

2.1 Mathematische Grundlagen

In diesem Abschnitt führen wir grundlegende mathematische Begriffe der Arbeit ein.

2.1.1 Menge

Eine *Menge* ist eine Ansammlung von „gleichartigen Objekten“. Ein „Objekt“ a mit der Eigenschaft $a \in A$ heißt *Element* der Menge A. Wir geben Mengen an, indem wir ihre Elemente aufzählen: $A = \{a_1, a_2, \ldots, a_n\}$. Oder wir geben eine Bedingung $p(a)$ an, die für die Elemente der Menge gilt: $A = \{a \mid p(a)\}$. Die *leere Menge* wird mit $\emptyset$ bezeichnet und mit $\emptyset = \{a \mid a \neq a\}$ charakterisiert, da die Bedingung $a \neq a$ von keinem Element erfüllt wird [Goo95]. Zuweilen schreiben wir auch $\{\}$ für die leere Menge, um den Mengenaspekt des entsprechenden Elementes hervorzuheben.

Wir definieren die Begriffe *Teilmenge* ($A \subseteq B$), *echte Teilmenge* ($A \subset B$), *Vereinigung* ($A \cup B$), *Durchschnitt* ($A \cap B$) und *Mengendifferenz* ($A \setminus B$) wie üblich [PS92]

$$A \subseteq B \quad \text{wenn aus } a \in A \text{ folgt } a \in B \tag{2.1a}$$

$$A \subset B \quad \text{wenn } A \subseteq B \text{ und } A \neq B \tag{2.1b}$$

$$A \cup B = \{a \mid a \in A \text{ oder } a \in B\} \tag{2.1c}$$

$$A \cap B = \{a \mid a \in A \text{ und } a \in B\} \tag{2.1d}$$

$$A \setminus B = \{a \mid a \in A \text{ und } a \notin B\} \tag{2.1e}$$

Die leere Menge ist Teilmenge jeder anderen nicht leeren Menge A ($\emptyset \subset A$). Zwei Mengen A und B heißen *disjunkt*, wenn gilt $A \cap B = \emptyset$.

Eine Menge kann auch Elemente enthalten, die wieder Mengen sind. Zu einer Menge A können wir die *Menge aller Teilmengen* (Potenzmenge) von A bilden. Wir bezeichnen die Menge aller Teilmengen einer Menge A mit $\mathcal{P}(A)$.

Die Menge der natürlichen Zahlen (einschließlich 0) bezeichnen wir mit $\mathbb{N}$. Und mit $\mathbb{B}$ bezeichnen wir die Menge der Wahrheitswerte wahr und falsch.

Wir definieren das *Produkt* von endlich vielen Mengen $A_1, A_2, \ldots, A_n$ $(n \in \mathbb{N})$ mit

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A_i \text{ für } i = 1, \ldots, n\} \tag{2.2}$$

Ein Element dieses Produktes heißt *n-Tupel*. Für ein i mit $1 \leq i \leq n$ heißt a_i die *i-te Komponente* des Tupels $(a_1, \ldots, a_n)$. Ein 2-Tupel nennen wir auch *Paar*. Für ein n-Tupel $a \in A_1 \times \cdots \times A_n$ und ein i mit $1 \leq i \leq n$ bezeichnet $a_{(i)}$ die i-te Komponente des Tupels a; wir nennen $a_{(i)}$ auch *Projektion* von a auf die i-te Komponente. Falls alle Mengen $A_1 = A_2 = \cdots = A_n = A$ gleich sind, schreiben wir statt $A \times \cdots \times A$ auch A^n. Außerdem gilt $A^1 = A$ und $A^0 = \emptyset$.

2.1.2 Relation und Abbildung

Eine *Relation* ist eine Teilmenge $\rho \subseteq A_1 \times \cdots \times A_n$. Eine *partielle Abbildung* oder *partielle Funktion* $f : A \to B$ ist eine Relation $f \subseteq A \times B$ so, dass aus $(a, b) \in f$ und $(a, c) \in f$ folgt $b = c$. Statt $(a, b) \in f$ schreiben wir $b = f(a)$. Der *Definitionsbereich* von f ist definiert mit

$$\text{def}_f = \{a \mid a \in A \text{ und es gibt ein } b \in B \text{ mit } b = f(a)\} \tag{2.3}$$

Der *Wertebereich* von f ist

$$\text{val}_f = \{b \mid b \in B \text{ und es gibt ein } a \in A \text{ mit } b = f(a)\} \tag{2.4}$$

Eine partielle Abbildung $f : A \to B$ mit $\text{def}_f = A$ heißt *vollständige Abbildung* oder kurz *Abbildung* bzw. *Funktion*. Definitions- und Wertebereich einer Funktion können selbst Produkte von Mengen sein. Für eine Funktion $f : A \times B \to C$ und ein Tupel $(a, b) \in A \times B$ schreiben wir $f(a, b)$ statt $f((a, b))$. Eine Funktion p mit dem Wertebereich $\mathbb{B}$ ($\text{val}_p = \mathbb{B}$) heißt *Prädikat*.

Für eine zweistellige Relation $\rho \subseteq A^2$ und Elemente $x, y \in A$ schreiben wir oft $x \rho y$ statt $(x, y) \in \rho$. Eine zweistellige Relation $\rho \subseteq A^2$ heißt *irreflexiv*, wenn es kein $x \in A$ mit $x \rho x$ gibt. Die Relation ρ heißt *transitiv*, wenn für alle $x, y, z \in A$ gilt aus $x \rho y$ und $y \rho z$ folgt $x \rho z$. Eine zweistellige Relation heißt *Halbordnung*, wenn sie irreflexiv und transitiv ist. Eine Halbordnung $\rho \subseteq A^2$ ist eine *Totalordnung*, wenn für alle $x, y \in A$ gilt entweder $x \rho y$ oder $y \rho x$ oder $x = y$ [Goo95].

2.1.3 Multimenge

Für Petrinetze werden häufig Multimengen benötigt, um auszudrücken, dass das gleiche Element mehrmals (in endlicher Anzahl) in einer Ansammlung von Elementen vorkommt. Beispielsweise kann auf einer Stelle eines Petrinetzes die gleiche Marke mehrmals vorkommen. Die Markierung einer Stelle ist häufig eine Multimenge von Marken.

Für eine Menge A nennen wir eine Abbildung $m : A \to \mathbb{N}$ eine *Multimenge* über A, wenn die Menge $\{a \mid a \in A \text{ und } m(a) \neq 0\}$ endlich ist. Für $a \in A$ ist $m(a)$ also die *Häufigkeit*, mit der a in der Multimenge m vorkommt. Die Menge aller Multimengen über einer endlichen Menge A bezeichnen wir mit $\mathcal{B}(A)$. Die *leere Multimenge* ist die Abbildung $[] : A \to \mathbb{N}$ mit $[](a) = 0$ für alle $a \in A$. Eine Multimenge geben wir auch so an $[a, a, \ldots, a, b, \ldots, b, \ldots]$, wobei jedes Element entsprechend seiner Häufigkeit wiederholt wird. Für Multimengen m_1 und m_2 ($m_1, m_2 \in \mathcal{B}(A)$) definieren wir den Begriff *Teilmenge* ($m_1 \sqsubseteq m_2$):

$$m_1 \sqsubseteq m_2 \quad \text{gdw. für alle } a \in A\text{:} \quad m_1(a) \leq m_2(a) \tag{2.5a}$$

Die *Vereinigung* ($m_1 \sqcup m_2$) und die *Multimengendifferenz* ($m_1 \boxminus m_2$) zweier Multimengen m_1 und m_2 ($m_1, m_2 \in \mathcal{B}(A)$) werden elementweise für alle $a \in A$ definiert:

$$(m_1 \sqcup m_2)(a) = m_1(a) + m_2(a) \tag{2.5b}$$

$$(m_1 \boxminus m_2)(a) = \begin{cases} m_1(a) - m_2(a) & \text{wenn } m_1(a) \geq m_2(a) \\ 0 & \text{sonst} \end{cases} \tag{2.5c}$$

Mit $|m|$ bezeichnen wir die *Kardinalität* einer Multimenge $m \in \mathcal{B}(A)$. Sie ist definiert mit

$$|m| = \sum_{a \in A} m(a) \tag{2.6}$$

Da eine Multimenge per Definition (siehe oben) immer endlich ist, ist die Kardinalität einer Multimenge immer eine natürliche Zahl. Für die Eigenschaft $m(a) \geq 1$ einer Multimenge $m \in \mathcal{B}(A)$ und eines Elementes $a \in A$ schreiben wir auch $a \in m$.

2.1.4 Sequenz

Wir nennen ein n-Tupel $a \in A^n$ auch eine *Sequenz* der Länge n über Elementen der Menge A. Die Menge A^* ist definiert durch

$$A^* = \bigcup_{i \in \mathbb{N}} A^i \tag{2.7}$$

und bezeichnet die Menge aller endlichen Sequenzen über der Menge A. Für die leere Sequenz (Sequenz der Länge 0) schreiben wir auch (), um ihren Tupelaspekt hervorzuheben.

Wir definieren die *Verkettung* von endlichen Sequenzen über A als Abbildung

$$\circ : A^* \times A^* \to A^* \tag{2.8a}$$

mit der Eigenschaft, dass für Sequenzen $a, b \in A^*$ mit $a \in A^n$ und $b \in A^m$ sei

$$a \circ b = \left(a_{(1)}, a_{(2)}, \ldots, a_{(n)}, b_{(1)}, \ldots, b_{(m)}\right) \tag{2.8b}$$

2.1.5 Algebra

Eine *Algebra* $\mathcal{A} = (B, O)$ besteht aus einer *Grundmenge* B und einer *Familie* von *Operationen* $O = \langle f_1, \ldots, f_n \rangle$ über B. Jede Operation f_i $(1 \leq i \leq n)$ ist eine Abbildung $f_i : B^{k_i} \to B$; k_i ist die *Stelligkeit* von f_i.

Wie üblich [EMC+99] unterscheiden wir eine nullstellige Operation $f : B^0 \to B$ nicht von einem gleichnamigen Element der Grundmenge B, das *Konstante* heißt. Beispielsweise schreiben wir $f \in B$ statt $f() \in B$. Außerdem verwenden wir manchmal Infixnotation (also $a \circ b$ statt $\circ(a, b)$) für 2-stellige Operationen.

In der vorliegenden Arbeit werden Algebren der Art $\mathcal{A} = (B, \langle \varepsilon, \oplus, \ominus \rangle)$ mit besonderen Eigenschaften verwendet. Sie werden benötigt, um Marken von Petrinetzen (siehe Abschn. 2.2) zu Markierungen „zusammenzusetzen“. Die Operationen dieser Algebren sind die folgenden Abbildungen

$$\begin{aligned} \varepsilon &: B^0 \to B \\ \oplus &: B^2 \to B \\ \ominus &: B^2 \to B \end{aligned} \tag{2.9}$$

Die Operationen $\oplus$ und $\ominus$ sind im Allgemeinen partiell; für alle $x, y \in B$ mit $x \neq \varepsilon$ und $y \neq \varepsilon$ sei

$$x \oplus y \neq x \tag{2.10a}$$
$$x \oplus y \neq y \tag{2.10b}$$
$$x \ominus y \neq x \tag{2.10c}$$

Die Teilalgebra $\mathcal{A}' = (B, \langle \varepsilon, \oplus \rangle)$ ist ein Monoid. Das heißt, es gelten die folgenden Eigenschaften für $x, y, z \in B$.

$$(x \oplus y) \oplus z = x \oplus (y \oplus z) \tag{2.11a}$$
$$x \oplus \varepsilon = \varepsilon \oplus x = x \tag{2.11b}$$

Das heißt, die Operation $\oplus$ ist eine assoziative Operation, und ε ist ein neutrales Element. Für die Operation $\ominus$ gilt

$$x \ominus \varepsilon = x \tag{2.11c}$$

$$x \ominus x = \varepsilon \tag{2.11d}$$

Weitere Eigenschaften der Operation $\ominus$ können wir zunächst nicht angeben, insbesondere gilt die Eigenschaft $(x \oplus y) \ominus y = x$ nicht für alle Algebren, die wir in dieser Arbeit verwenden (siehe Abschn. 3.2.2). Wir nennen eine Algebra, die wie oben in den Gleichungen (2.9) bis (2.11) charakterisiert ist, eine *Markierungsalgebra*.

In dieser Arbeit verwenden wir *Klassen von Markierungsalgebren*. Eine solche Klasse $\mathfrak{A}$ konstruiert aus einer beliebigen Menge A mit Hilfe einer Abbildung $u : A \to B$, mit $u(a) \neq \varepsilon$ für alle $a \in A$, eine Markierungsalgebra $\mathcal{A} = (B, \langle \varepsilon, \oplus, \ominus \rangle)$ (ähnlich einer freien Konstruktion einer Algebra [EMC+99]). Für alle $a \in A$ gibt es ein $b \in B$ so, dass $b = u(a)$. Für $x, y \in A$ mit $x \neq y$ sei $u(x) \neq u(y)$. Für die Konstruktion der Markierungsalgebra $\mathcal{A}$ aus der Menge A schreiben wir $\mathfrak{A}(A) = \mathcal{A}$. Eine so konstruierte Algebra $\mathcal{A}$ nennen wir auch *Algebra der Klasse* $\mathfrak{A}$, und wir schreiben dafür auch $\mathcal{A} \in \mathfrak{A}$. Für alle Klassen von Markierungsalgebren gelten die Operationsdefinitionen (2.9) sowie die Eigenschaften (2.10) und (2.11). Wir fordern, dass die Abbildung u für alle Elemente der Menge A atomare Elemente der Menge B erzeugt. Das heißt, die Elemente von B, die mit u aus A hervorgehen, sind unteilbar; für alle $a \in A$ und für alle $x, y \in B$ mit $x \neq \varepsilon$ und $y \neq \varepsilon$ sei

$$x \oplus y \neq u(a) \tag{2.12a}$$

$$u(a) \ominus x \text{ nicht definiert für } x \neq u(a) \text{ und } x \neq \varepsilon \tag{2.12b}$$

Atomare Elemente, aus denen die Elemente der Menge B zusammengesetzt werden, können nicht verschwinden bzw. auftauchen.

Für ein Element $b \in B$ schreiben wir auch $b \in \mathfrak{A}(A)$. Ebenso benutzen wir die Algebra $\mathfrak{A}(A)$ auch an Stelle ihrer Grundmenge B.

2.2 Petrinetze und Petrinetz-Typen

In diesem Abschnitt werden Petrinetze mit Hilfe von Beispielen erklärt. Zunächst werden allgemeine Begriffe der Petrinetze am Beispiel eines S/T-Netzes (Abschn. 2.2.1) eingeführt. Dann werden verschiedene Petrinetz-Typen und die Unterschiede zwischen ihnen erläutert (Abschn. 2.2.2). Und schließlich werden im Abschn. 2.2.3 bisherige Klassifikationen von Petrinetz-Typen diskutiert.

2.2.1 Petrinetze im Beispiel

Abbildung 2.1 zeigt die grafische Darstellung eines Petrinetzes. Die formale Definition

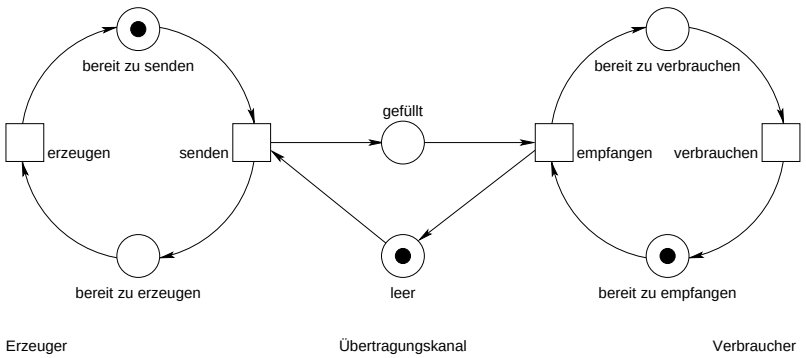


Abbildung 2.1: Ein System eines Erzeugers und eines Verbrauchers als Petrinetz N_1

folgt in Kap. 3. Mit diesem Petrinetz wird ein einfaches Erzeuger/Verbraucher-System modelliert. Im linken Teil der Abb. 2.1 werden Aktionen und Zustände eines Erzeugers und im rechten Teil die eines Verbrauchers dargestellt. Zwischen Erzeuger und Verbraucher gibt es einen Übertragungskanal. Ein Erzeuger erzeugt ein (im Petrinetz *nicht* dargestelltes) Produkt, das über einen Übertragungskanal an einen Verbraucher gesendet wird. Der Verbraucher empfängt das Produkt und verbraucht es. Der Erzeuger wechselt zwischen seinen Zuständen bereit zu erzeugen und bereit zu senden. Er kann die Aktionen erzeugen und senden ausführen. Der Übertragungskanal ist an den Aktionen senden (des Erzeugers) und empfangen (des Verbrauchers) beteiligt. Sein Zustand ist leer bzw. gefüllt. Der Verbraucher wechselt zwischen seinen Zuständen bereit zu verbrauchen und bereit zu empfangen. Er kann die Aktionen empfangen und verbrauchen ausführen. Der Systemzustand des gesamten Systems wird durch die in Abb. 2.1 dargestellten Vollkreise modelliert. Die Abbildung des Petrinetzes stellt den Zustand dar, in dem der Erzeuger bereit zu senden ist, der Übertragungskanal leer ist, und der Verbraucher bereit zu empfangen ist. Die einzige Aktion, die in dem System ausgeführt werden kann, ist die Aktion senden, die den Erzeuger in den Zustand bereit zu erzeugen und den Übertragungskanal in den Zustand gefüllt versetzt.

Ein Petrinetz besteht aus *Stellen*, *Transitionen* und *Kanten*. Zusammenfassend bezeichnen wir die Elemente aller drei Arten als *Netzelemente*. Stellen und Transitionen fassen wir auch mit dem Begriff *Knoten* zusammen. Eine Stelle wird grafisch durch einen Kreis oder eine Ellipse dargestellt. Sie repräsentiert den statischen (unveränderlichen) Aspekt eines lokalen Systemzustandes. Eine Transition modelliert lokale Zustandsübergänge. Sie wird grafisch durch ein Quadrat bzw. Rechteck dargestellt. Welche lokalen Zustände ineinander überführt werden, wird mit Kanten modelliert. Eine Kante verläuft von einer Stelle zu einer Transition bzw. umgekehrt. Die Richtung der Kante wird grafisch durch einen Pfeil dargestellt. Für solche Kanten kann die Menge aller Vor- bzw. Nachknoten eines ausgewählten Knotens bestimmt wer-

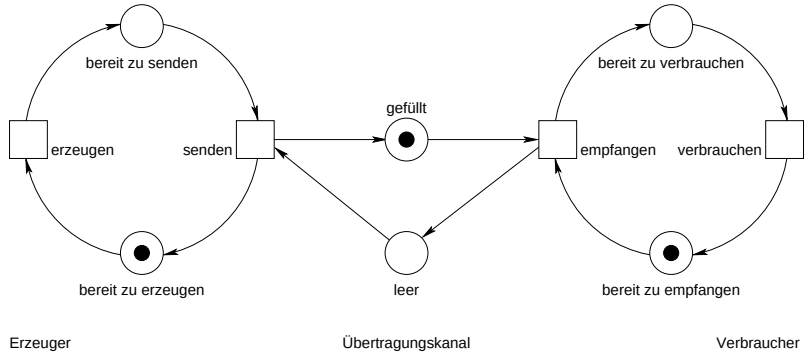


Abbildung 2.2: Petrinetz N_1 nach dem Schalten der Transition senden

den. Die Vorplätze einer Transition sind die Stellen, von denen zu der Transition eine Kante führt. Die Nachplätze einer Transition sind dann die Stellen, zu denen von der Transition eine Kante führt. Für Vor- und Nachtransitionen einer Stelle gilt entsprechendes.

Außerdem werden in einem Petrinetz den lokalen Systemzuständen dynamische Aspekte zugeordnet. Sie werden durch Marken auf Stellen modelliert. In unserem Beispiel in Abb. 2.1 werden Marken grafisch durch schwarz gefüllte Kreise in Stellen dargestellt. Die Richtung einer Kante in einem Petrinetz zeigt an, von welchen Stellen eine Transition Marken konsumiert bzw. auf welchen sie Marken produziert. Eine Transition, die auf ihren Vorplätzen genügend Marken vorfindet ist *konzessioniert*. In vielen Petrinetzen reicht diese Bedingung noch nicht aus, damit eine Transition schalten kann. Wenn dagegen *alle* Bedingungen erfüllt sind, heißt die Transition *aktiviert*. Der Begriff der Aktivierung einer Transition schließt damit den Begriff der Konzession ein.

Eine aktivierte Transition kann schalten. Dabei konsumiert sie Marken von ihren Vorplätzen und produziert Marken auf ihren Nachplätzen. Wir sprechen auch vom *Fließen* einer Marke (bzw. *Markenfluß*), wenn sie konsumiert bzw. produziert wird. In unserem Beispiel in Abb. 2.1 ist genau eine Transition aktiviert, die mit der Bezeichnung senden, da sie die einzige Transition des Netzes ist, deren Vorplätze alle markiert sind. Wenn diese Transition schaltet, entsteht das Petrinetz in Abb. 2.2 mit der dort gezeigten Markenverteilung.

Im Allgemeinen sieht man in der Petrinetz-Theorie den Graphen aus Stellen, Transitionen und Kanten als statisch und unveränderlich an und beschreibt die Verteilung von Marken als Abbildung der Stellen in eine Menge von Markierungen. Eine *Markierung* ist also eine gewisse Anzahl und Struktur von Marken auf einer Stelle. Die Verteilung aller Marken in einem Netz bezeichnen wir als *Markierung des Netzes*.

2.2.2 Petrinetz-Typen

Im vorangegangenen Abschnitt haben wir die wichtigsten Begriffe eines Petrinetzes am Beispiel kennen gelernt. Das darin verwendete Netz ist vom Petrinetz-Typ „Stellen-Transitions-Netz“. Der Graph eines Netzes, egal von welchem Typ, besteht aus Stellen, Transitionen und Kanten. Aber welche Marken in einem Netz verwendet werden, wie sie zu Markierungen zusammengesetzt werden, welche Anschriften an Netzelementen vorkommen und wie Transitionen genau schalten – also die Schaltregel des Netzes – ist abhängig vom Petrinetz-Typ, zu dem das Netz gehört.

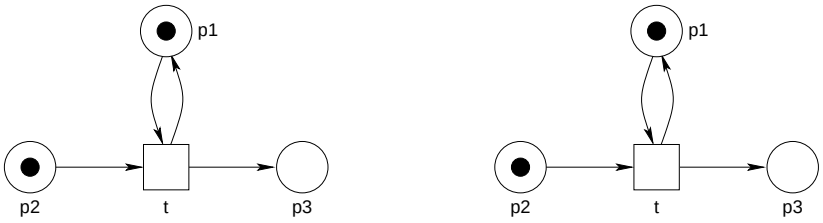
Die Anschriften an Netzelementen (bzw. wo nötig am Netz) fassen wir mit dem Begriff *Label* zusammen. Einige Label wie der Name bzw. eineindeutige Bezeichner eines Netzelementes haben für die meisten Petrinetz-Typen keine Semantik. Wir werden sie deshalb in einem strengen Sinne nicht zu einem Petrinetz-Typ zählen. Stattdessen können solche syntaktischen Label zu jedem Petrinetz-Typ hinzugefügt werden, ohne das Schaltverhalten zu verändern. Syntaktische Label haben vor allem Bedeutung, um die syntaktische Korrektheit von (*high level*) Petrinetzen zu überprüfen. Auch die häufig besonders in Petrinetz-Werkzeugen verwendete initiale Markierung, die die Verteilung von Marken in einem Anfangszustand modelliert, ist ein syntaktisches Label.

Im Folgenden wollen wir einige Petrinetz-Typen beschreiben, indem wir die Markierung, Label an Netzelementen oder dem Netz und die Schaltregel ihrer Netze charakterisieren. Wir wollen damit einen Eindruck von der Welt der Petrinetz-Typen erhalten.

Stellen-Transitions-Netze

Ein *Stellen-Transitions-Netz* [Rei87, Lau87] (abgekürzt: S/T-Netz) hat nicht unterscheidbare Marken, die meist als *schwarze Marken* (*black tokens*) bezeichnet werden und grafisch durch schwarze Vollkreise dargestellt werden. Auf einer Stelle eines S/T-Netzes dürfen beliebig viele Marken vorkommen. Grafisch stellt man eine große Anzahl von Marken durch die entsprechende Zahl dar.

Eine Transition in S/T-Netzen konsumiert von ihren Vorplätzen beim Schalten so viele Marken, wie die Inschrift an der Kante anzeigt, die die Transition mit ihrem Vorplatz verbindet. Ebenso zeigt die Kanteninschrift einer Kante zu einem Nachplatz, wie viele Marken eine Transition beim Schalten auf diesem Nachplatz produziert. In S/T-Netzen ist die Kanteninschrift eine natürliche Zahl. Üblicherweise wird eine Kanteninschrift „1“ nicht explizit dargestellt, da schon die Existenz der Kante das Fließen wenigstens einer Marke beim Schalten der entsprechenden Transition impliziert.



(a) Elementares Netzsystem (b) sicheres S/T-Netz

Abbildung 2.3: Gleiche grafische Darstellung für Petrinetze verschiedener Typen

Elementare Netzsysteme

Wie S/T-Netze gehören auch *Elementare Netzsysteme* [Thi87, Roz87] zu den so genannten *low level* Netzen. *Low level* nennt man alle Petrinetz-Typen, die nur eine Art von (ununterscheidbaren) Marken für ihre Petrinetze zulassen.

In Elementaren Netzsystemen darf auf einer Stelle immer nur höchstens eine Marke vorkommen. Das heißt, eine Transition konsumiert von ihren Vorplätzen beim Schalten jeweils eine Marke und produziert auf ihren Nachplätzen jeweils nur eine Marke. Eine Transition ist nur dann aktiviert, wenn ihre Vorplätze jeweils eine Marke enthalten und ihre Nachplätze keine Marken enthalten. Die Schaltregel von Elementaren Netzsystemen unterscheidet sich also etwas von der Schaltregel der S/T-Netze. Da über eine Kante beim Schalten in einem Elementaren Netzsystem jeweils nur eine Marke fließt, ist eine Kanteninschrift für diesen Petrinetz-Typ nicht definiert.

Auch in so genannten sicheren S/T-Netzen [Sta90] kommt auf den Stellen immer höchstens eine Marke vor und alle Kanteninschriften sind „1“. Dennoch unterscheiden sich sichere S/T-Netze von Elementaren Netzsystemen fundamental, wie die Beispiele in Abb. 2.3 zeigen. Während die Transition t in Abb. 2.3(a) des Elementaren Netzsystems nicht schalten kann, da auf ihrem Nachplatz p1 bereits eine Marke vorhanden ist, ist die Transition t in Abb. 2.3(b) aktiviert, da die Schaltregel für S/T-Netze lediglich das Vorhandensein von Marken in ausreichender Anzahl auf den Vorplätzen einer Transition erfordert. Nach dem Schalten der Transition t in Abb. 2.3(b) liegt auf der Stelle p1 eine Marke, da t auf p1 sowohl eine Marke konsumiert als auch produziert. Die Stellen p2 und p3 verlieren bzw. gewinnen je eine Marke.

Coloured Petri Nets

Coloured Petri Nets (auch: gefärbte Petrinetze) wurden von Kurt Jensen eingeführt [Jen83, Jen92]. Dieser Petrinetz-Typ ist ein so genannter *high level* Petrinetz-Typ (vgl. auch [Gen87]), da die Marken in *Coloured Petri Nets* voneinander unterscheidbar sein können bzw. verschiedene Farben haben. Abbildung 2.4 zeigt ein Beispiel für ein

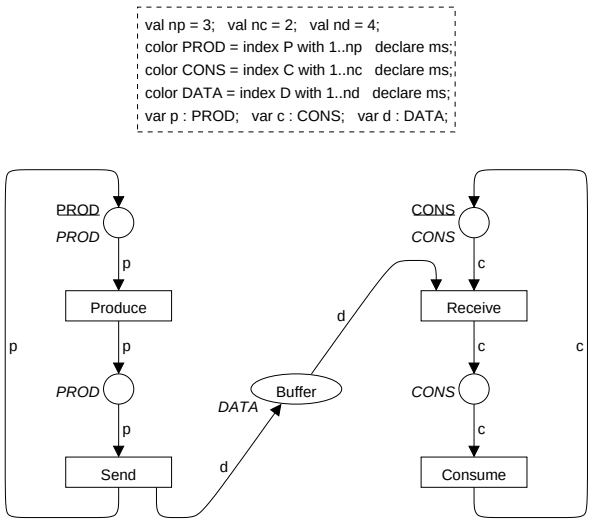


Abbildung 2.4: Ein *Coloured Petri Net* (aus [Jen92])

Coloured Petri Net in der für sie definierten Syntax [Jen92]. Es hat einige Label mehr, als die bis hierher vorgestellten Petrinetz-Typen.

So wird eine Stelle mit einem Label *Colour* (in Abb. 2.4 z. B. *PROD*) ergänzt. Kanten erhalten eine Kanteninschrift (z. B. p in Abb. 2.4) und Knoten einen Bezeichner, der grafisch innerhalb des Knotens dargestellt wird (z. B. Buffer). Das Netz erhält ein Label, *Declaration* genannt. Dieses Label deklariert Sorten (color), Konstanten (val) und Variablen (var), die im Netz als *Colour*-Label bzw. in Kanteninschriften verwendet werden. Die initiale Markierung einer Stelle wird neben der Stelle unterstrichen dargestellt. Sie ist ein Ausdruck, der in eine Multimenge ausgewertet wird. Außerdem kann eine Transition ein Label *transition guard* für eine Aktivierungsbedingung erhalten. Die Aktivierungsbedingung wird ausgewertet, wenn die Konzession ihrer Transition berechnet wird.

Der Schaltmodus einer Transition belegt alle freien Variablen in der Aktivierungsbedingung der Transition und in den Kanten, die in der Transition enden bzw. beginnen mit Marken der entsprechenden Sorte, wobei gleiche Variablen gleiche Werte erhalten. Eine Transition ist mit diesem Schaltmodus aktiviert, wenn auf den Vorplätzen der Transition die Marken der entsprechend ausgewerteten Kanteninschriften enthalten sind und wenn die Aktivierungsbedingung der Transition für diesen Schaltmodus gültig ist. Beim Schalten der Transition in diesem Schaltmodus werden die entsprechenden Marken von den Vorplätzen konsumiert und auf den Nachplätzen produziert.

Algebraische Petrinetze

Algebraische Petrinetze, (z. B. [Rei91b, Rei98]) sind ebenfalls *high level* Petrinetze. Sie werden vor allem eingesetzt, um Netzwerkalgorithmen und andere verteilte Algorithmen zu modellieren und vor allem zu verifizieren. Wir unterscheiden für Algebraische Petrinetze Schemata und Instanzen. Ein Schema beschreibt einen Algorithmus für eine Klasse von Konfigurationen, wobei Anforderungen an die Konfiguration mit Hilfe einer Algebra spezifiziert werden. In der Verifikation eines Schemas werden die Eigenschaften der Algebra ausgenutzt. Eine Instanz dagegen beschreibt eine bestimmte Konfiguration des Algorithmus. Instanzen von Algebraischen Petrinetzen sind *Coloured Petri Nets* ähnlich. Für sie gilt eine vergleichbare Schaltregel wie oben.

Algebraische Petrinetze führen weitere syntaktische Label ein wie externe Transitionen und faire Transitionen, die in der Verifikation ausgenutzt werden und notwendige Annahmen über den Algorithmus modellieren. Eine externe Transition drückt aus, dass ein verteilter Algorithmus in ein umfassenderes System eingebunden ist: Ein endlicher Ablauf des Algebraischen Petrinetzes darf keine Transitionen aktivieren, die nicht extern sind. Eine faire Transition charakterisiert die Abläufe eines Algebraischen Petrinetzes als unfair, in denen sie immer wieder aktiviert ist, aber nie schaltet. Beide Label haben mit der Schaltregel nichts zu tun.

DAWN-Netze [WWV+97] sind eine Weiterentwicklung Algebraischer Petrinetze. In DAWN-Netzen werden faire Transitionen durch faire Kanten ersetzt, da die Fairness-Annahme, die zur Modellierung verteilter Systeme verwendet wird, auf die Verwendung bestimmter Ressourcen (die mit einer Stelle und Marken modelliert werden) zurückgeführt werden kann.

Signal-Netzsysteme

Signal-Netzsysteme [Roc01] definieren die Semantik von Hans-Michael Hanischs *Condition/Event-Systems* [HL00]. Diese *Condition/Event-Systems* verbinden Petrinetze mit der traditionellen Modellierungstechnik der Steuer- und Regelungstechnik. Der Ansatz dieser Modellierung beruht auf der Kombination von Modulen. Abbildung 2.5 zeigt ein Beispiel, wie Module kombiniert werden. Dabei werden zwischen

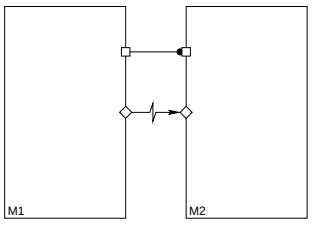


Abbildung 2.5: Modellieren mit Modulen

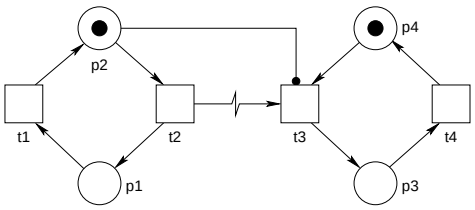


Abbildung 2.6: Signal-Netzsystem (zwei Transitionen aktiviert)

Modulen zwei Arten von Signalen – Bedingungs- bzw. Ereignissignal genannt – gesendet. Jedes Signal verbindet jeweils passende so genannte *ports*. Jeder *port* ist entweder Ursprung oder Ziel eines Signals. Ein Ereignissignal ist ein Impuls und wird deshalb mit einem blitzartigen Pfeil dargestellt. Ein Bedingungssignal ist in gewisser Weise permanent. Es wird durch eine gerichtete Kante mit einem Kreis dargestellt.

Signal-Netzsysteme definieren nun die Semantik jedes Moduls und damit eines gesamten *Condition/Event-Systems*. Sie verbinden damit die mächtigen Möglichkeiten der Analyse von Petrinetzen mit der herkömmlichen Modellierungstechnik. Auf höherer Ebene kombiniert der Modellierer weiterhin auf gewohnte Weise Module (wie in Abb. 2.5 gezeigt). Und die Theorie [HL00, Roc01] sorgt für korrekte Kombinationen und die Analyse des Gesamtsystems. Das Werkzeug SENA [Roc01] bietet dafür die software-technische Realisierung.

Ein Signal-Netzsystem besteht wie ein S/T-Netz aus Stellen, Transitionen und Kanten sowie ununterscheidbaren Marken. Außerdem hat ein Signal-Netzsystem zwei Arten von Signalen, die mit weiteren Kanten modelliert werden. Bedingungssignale sind permanente Signale, die verarbeitet werden können, wenn sie benötigt werden; während Ereignissignale Impulssignale sind, die nur verarbeitet werden können, wenn sie aufgefangen werden.

Abbildung 2.6 zeigt ein Beispiel für ein Signal-Netzsystem. Ein Bedingungssignal wird in Signal-Netzsystemen mit einer *Testkante* modelliert. Das ist eine Kante zwischen einer Stelle und einer Transition mit einer besonderen Semantik. Eine Transition, die mit einer Stelle über eine solche Kante verbunden ist, benötigt die entsprechenden Marken auf der Stelle zu ihrer Konzession. Beim Schalten der Transition fließt über diese Kante jedoch keine Marke (vgl. Abschn. 3.4). Bedingungssignale werden grafisch durch einen kleinen Vollkreis an der empfangenden Transition dargestellt, wie die Kante zwischen p2 und t3 in Abb. 2.6.

Ein Ereignissignal wird mit einer *Signalkante* modelliert. Dies ist eine Kante von einer Transition zu einer anderen. Eine Transition, die das Ziel einer solchen Kante ist, kann nur dann schalten, wenn sie konzessioniert ist und über alle Signalkanten, die zu ihr führen, jeweils ein Signal empfängt. Sie empfängt ein Signal, wenn die Signal aussendende Transition schaltet. Eine Transition, zu der keine Signalkante führt, wird *spontane Transition* genannt, da ihr Schalten lediglich von ihrer Konzession abhängt.

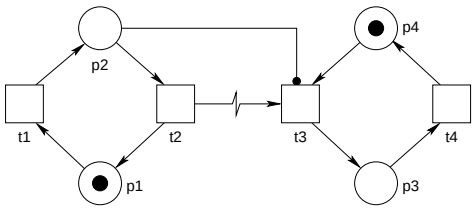


Abbildung 2.7: Signal-Netzsystem (eine Transition aktiviert)

Grafisch werden Ereignissignale durch einen blitzartigen Pfeil dargestellt, wie der Pfeil von t2 nach t3 in Abb. 2.6.

In unserem Beispiel aus Abb. 2.6 sind die Transitionen t2 und t3 konzessioniert, denn der Vorplatz von t2 ist ebenso markiert wie der Vorplatz von t3 und das Bedingungssignal zu t3 liegt an, da p2 markiert ist. Die Transition t2 kann *spontan* schalten, da sie keine Ereignissignale empfängt; wohingegen t3 nur schalten kann, wenn t2 – der Ursprung des benötigten Ereignissignals – schaltet. In dem Beispiel aus Abb. 2.7 kann dagegen nur die spontane Transition t1 schalten, denn die konzessionierte Transition t3 empfängt kein Signal, da die Transition t2 nicht konzessioniert ist und demzufolge nicht schaltet.

Objekt-Petrinetze

Eine besondere Form von *high level* Netzen stellen *Objekt-Petrinetze* [Val98] dar, eingeführt von Rüdiger Valk. Ein Objekt-Petrinetz besteht aus einem Systemnetz und mehreren Objektnetzen. Objekt-Petrinetze werden verwendet, um sowohl das System mit seinen Ressourcen als auch Aufgaben des Systems zu modellieren. Systemnetze modellieren beispielweise eine Maschinenstrecke und die Objektnetze die Werkstücke, die in der Maschinenstrecke verschieden voneinander bearbeitet werden. Ein anderes Beispiel ist eine Abteilung (Systemnetz), die verschiedene Geschäftsvorfälle (Objektnetze) je nach deren Anforderung bearbeitet.

Jedes Objektnetz ist ein Elementares Netzsystem wie oben beschrieben. Jedes Objektnetz ist außerdem eine Marke im Systemnetz. Das Systemnetz ist ein *high level* Netz. Seine Marken sind Objektnetze oder schwarze Marken, wobei auf einer Stelle nicht beide Arten von Marken vorkommen dürfen. Eine Kanteninschrift in einem Systemnetz ist entweder (wie für S/T-Netze) eine natürliche Zahl, wenn die entsprechende Stelle schwarze Marken enthält, oder eine Menge von Objektnetz-Bezeichnern, wenn die entsprechende Stelle Objektnetze als Marken enthält. Jede Transition des Systemnetzes, die Objektnetze schaltet, muss ein Objektnetz von einem Vorplatz zu genau einem Nachplatz fließen lassen. Bei jedem Schalten einer Transition wird höchstens ein Objektnetz bewegt. Außerdem haben manche Transitionen des Systemnetzes und der Objektnetze ein zusätzliches Label, *Interaktion* genannt, das modelliert, dass die

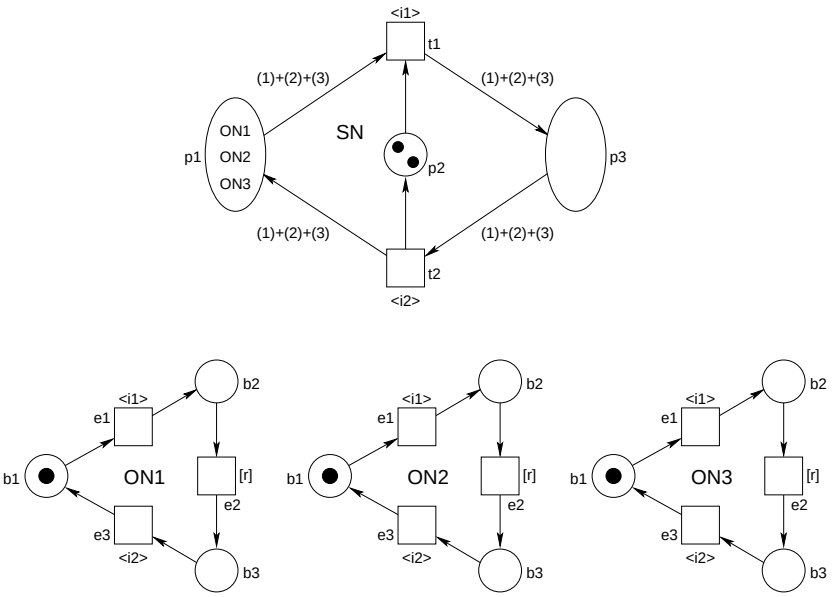


Abbildung 2.8: Objekt-Petrinetz N_O (aus [Val98])

Transitionen, die die gleiche Interaktion haben, synchron schalten und nur dann, wenn sie jeweils konzessioniert sind. Es gibt zwei Arten von Interaktionen, eine System-/Objektnetz-Interaktion und eine Objekt-/Objektnetz-Interaktion. Beide Arten von Interaktion funktionieren im Sinne eines *handshake*.

Abbildung 2.8 zeigt ein Beispiel für ein Objekt-Petrinetz. Es besteht aus dem Systemnetz SN und den Objektnetzen ON1, ON2 und ON3. Die Transition t1 des Systemnetzes ist konzessioniert mit einem Objektnetz von Stelle p1 und einer schwarzen Marke von Stelle p2. Wenn die Transition t1 schaltet, fließt ein Objektnetz von p1 zur Stelle p3, und von Stelle p2 wird eine schwarze Marke konsumiert. Die Transition t1 interagiert mit der Transition e1 desjenigen Objektnetzes, das als Marke von t1 konsumiert wird, da beide das Label der System-/Objektnetz-Interaktion <i1> haben. Wenn ON1 dieses Objektnetz ist, dann verändert es also seinen Zustand, so dass nun seine Stelle b2 markiert ist.

Wenn zwei Objektnetze als Marke auf einer Stelle des Systemnetzes sind, können sie synchron schalten, wenn ihre Transition, die das selbe Label der Objekt-/Objektnetz-Interaktion haben, konzessioniert sind. In Abb. 2.9 ist ein Beispiel für solch eine Situation zu sehen. In den Objektnetzen ON1 und ON2 auf der Stelle p3 des Systemnetzes SN sind jeweils die Transitionen e2 konzessioniert. Sie schalten synchron, da beide das Label [r] tragen. Nach dem Schalten sind jeweils die Stellen b3 der Objektnetze markiert, die als Marken nach wie vor auf der Stelle p3 von SN liegen. Beide

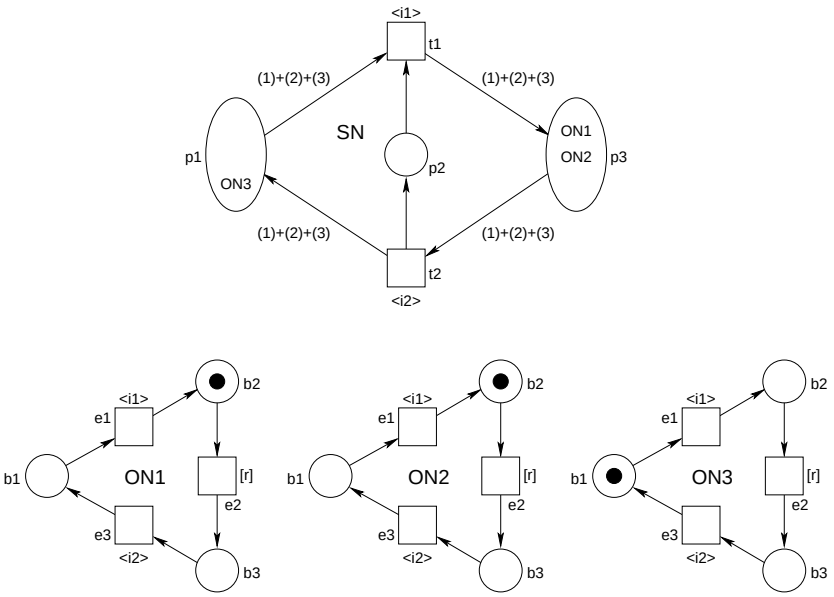


Abbildung 2.9: Objekt-Petrinetz N_O in einem anderen Zustand

Objektnetze können jeweils über das synchrone Schalten von `t2` und `e3` einen Zustand erreichen, in dem sie als Marke auf `p1` liegen und in dem `b1` markiert ist.

Weitere vergleichbare Ansätze sind *Objekt-Orientierte Petrinetze* [EJN97] und *Referenznetze* [KMW99, KWD01]. Eine Marke eines Referenznetzes ist eine Referenz auf ein Objekt eines bestimmten Datentyps. Eine Referenz kann dupliziert werden. Ein Referenznetz ist ein Objekt des Datentyps Referenznetze. Eine Referenz auf ein Objekt der Referenznetze kann eine Marke eines Referenznetzes sein. Das Werkzeug Renew [Ren99] ist Modellierungs- und Simulationswerkzeug für Referenznetze. Es ist in Java implementiert und benutzt daher neben Referenzen auf Netze auch Referenzen auf Java-Objekte als Marken für Referenznetze.

Zeit-Petrinetze

Als *Zeit-Petrinetze* wollen wir die Petrinetze bezeichnen, die Uhren und diesbezügliche Zeitanschriften verwenden. Zeit-Petrinetze wurden von Peter Starke klassifiziert [Sta95]. Zeitkonzepte wurden zunächst für *low level* Netze eingeführt, lassen sich aber relativ einfach für *high level* Netze anpassen [Jen92], da sie weitgehend unabhängig von Petrinetz-Konzepten sind. Stellvertretend für Zeit-Petrinetze beschreiben wir hier *Timed Petri Nets* [Ram74]. *Timed Petri Nets* bestehen aus S/T-Netzen mit Zeitanschriften (als Label) an Transitionen. Jede Transition hat eine Uhr, die (bei 0) gestartet wird, wenn die Transition konzessioniert ist (im Sinne eines S/T-Netzes;

also genügend Marken auf ihren Vorplätzen vorfindet). Eine Transition kann schalten, wenn sie weiterhin konzessioniert ist und ihre Zeitanschrift kleiner oder gleich der Stellung ihrer Uhr ist. Das heißt, eine Transition schaltet nach einer gewissen Präsenzzeit der zum Schalten nötigen Marken.

Eine Uhr wird ausgeschaltet, wenn die Transition, ihre Konzession verliert. *Timed Petri Nets* werden verwendet, um zeitliche Aspekte in Systemen zu modellieren und vor allem zu analysieren.

Stochastische Petrinetze

Als *Stochastische Petrinetze* werden im Allgemeinen Zeit-Petrinetze bezeichnet, die eine exponentiell verteilte Funktion als Zeitanschrift zulassen. Damit können Prozesse simuliert und analysiert werden, in denen eine als Zufallsgröße angenommene Bearbeitungszeit eine Rolle spielt. Im Besonderen werden die Netze als Stochastische Petrinetze [AMC87] bezeichnet, die *Timed Petri Nets* um eine exponentielle Verteilungsfunktion als Zeitanschrift an Transitionen erweitern. Eine Transition schaltet dann, wenn sie konzessioniert ist und die Präsenzzeit der entsprechenden Marken auf den Vorplätzen für einen aktuellen Wert der Verteilungsfunktion erreicht ist.

Kein Petrinetz-Typ: Zero Safe Nets

Zero Safe Nets wurden von Roberto Bruni und Ugo Montanari eingeführt [BMoo]. *Zero Safe Nets* sind S/T-Netzen vergleichbar. Sie sind jedoch im Gegensatz zu allen bisher vorgestellten Petrinetz-Typen in diesem Abschnitt *kein* Petrinetz-Typ.

Zero Safe Nets haben zusätzlich besondere Stellen – *zero places*. Diese *zero places* modellieren instabile bzw. ungewünschte Zwischenzustände eines Systems. Nur stabile Zustände sind gültige Systemzustände. Eine Transition eines *Zero Safe Nets* kann in einem stabilen Zustand nur schalten, wenn sie Bestandteil einer Folge von Transitionen ist. Eine Schaltfolge von Transitionen wird ausgeführt, wenn durch einige Transitionen dieser Folge markierte *zero places* ihre Marken durch andere Transitionen der Folge wieder verlieren.

Abbildung 2.10 zeigt ein Beispiel für ein *Zero Safe Net* mit den *zero places* z1 bis z4. Die Transitionen t2 und t3 sind nebenläufig zueinander konzessioniert. Sie können jedoch nicht in einer Folge von Transitionen schalten, da dann die *zero places* z2 und z3 markiert wären, woraufhin keine weitere Transition schalten kann. Die einzigen Folgen von Transitionen, die im Netz in Abb. 2.10 schalten können sind t1, t3, t5 bzw. t2, t4, t6, die zu einer Marke auf p3 bzw. p4 führen.

Aufgrund dieser Schaltregel, die das Ergebnis des Schaltens einer Transition (also die von ihr erzeugte Markierung) zur Voraussetzung des Schaltens anderer Transitionen macht, um einen Zustandsübergang zu modellieren, wollen wir *Zero Safe Nets* nicht als Petrinetz-Typen auffassen. Im Gegensatz dazu setzen z. B. Sig-

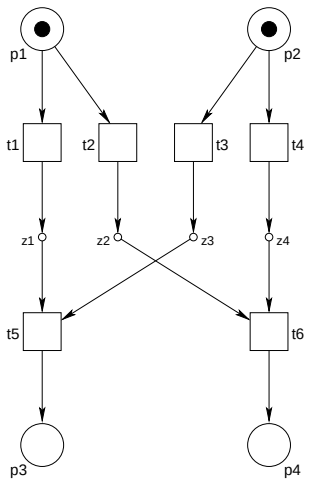


Abbildung 2.10: Ein *Zero Safe Net* (aus [BMoo])

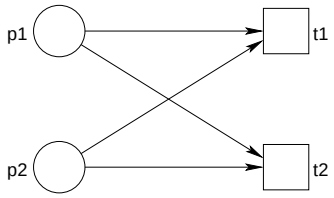
nal-Netzsysteme lediglich *das Schalten* einiger Transitionen für das Schalten anderer Transitionen voraus, aber nicht das Ergebnis. In Abschn. 3.5.6 wird dies näher begründet.

Petrinetz-Typ vs. Eigenschaft eines Petrinetzes

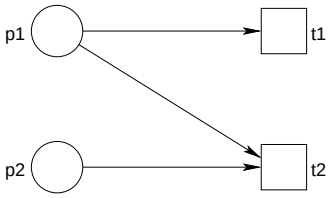
In diesem Abschnitt haben wir bisher Beispiele für Petrinetz-Typen betrachtet. Verschiedentlich wird für den Begriff Petrinetz-Typ auch der Begriff *Netzklasse* verwendet. Wir wählen den Begriff Petrinetz-Typ in Abgrenzung zu den Netzklassen, die strukturelle Eigenschaften von Petrinetzen hervorheben und meistens unabhängig vom Petrinetz-Typ im Sinne der vorliegenden Arbeit sind.

Ein Beispiel einer solchen Netzklasse ist die Klasse der zusammenhängenden Netze. Die Eigenschaft Zusammenhang ist eine Eigenschaft des dem Petrinetz zugrundeliegenden Graphen. Danach gehören alle die Petrinetze zur Klasse der zusammenhängenden Petrinetze in denen jeder Knoten von jedem anderen über beliebige Kanten erreichbar ist. Das Netz in Abb. 2.1 ist z. B. zusammenhängend. Ob das Signal-Netzsystem in Abb. 2.6 zusammenhängend ist, hängt dagegen von der genauen Definition der Eigenschaft Zusammenhang für Signal-Netzsysteme ab. Für jeden oben vorgestellten Petrinetz-Typ (und auch für *Zero Safe Nets*) kann jeweils die Klasse der zusammenhängenden Netze definiert werden.

Ein weiteres Beispiel für eine Netzklasse, die kein Petrinetz-Typ im Sinne dieser Arbeit ist, ist die Klasse der *extended free choice* Netze [Bes87, DE95]. Die Eigenschaft *extended free choice* ist eine strukturelle Eigenschaft von Petrinetzen. Ein Petrinetz hat diese Eigenschaft, wenn jede Nachtransition einer Stelle die selben Vorplätze hat.



(a) Petrinetz mit der Eigenschaft *extended free choice*



(b) Petrinetz ohne die Eigenschaft *extended free choice*

Abbildung 2.11: *Extended free choice*

Die Abb. 2.11(a) zeigt ein Beispiel für ein Netz mit der Eigenschaft *extended free choice*. Wohingegen das Netz in Abb. 2.11(b) diese Eigenschaft nicht hat, da t1 als Nachtransition von p1 nicht die selben Vorplätze hat wie t2 – eine weitere Nachtransition von p1. Auch diese Eigenschaft ist eine strukturelle Eigenschaft von Petrinetzen und kann für beliebige Petrinetz-Typen definiert werden.

Eine weitere Eigenschaft von Petrinetzen ist die weiter oben beschriebene Sicherheit in S/T-Netzen [Sta90]. Dies ist eine Eigenschaft, die von der Schaltregel abhängt. Sie bildet ebenfalls keinen Petrinetz-Typ im Sinne dieser Arbeit. Ein S/T-Netz ist sicher, wenn es keine Markierung des Netzes gibt, die von der Anfangsmarkierung des Netzes erreichbar ist, also durch Schalten von Transitionen erzeugt wird, und in der sich auf einer Stelle mehr als eine Marke befindet. Die Eigenschaft der Sicherheit von S/T-Netzen ist eine dynamische Eigenschaft, da sie von der Schaltregel und der Anfangsmarkierung abhängt und sich auf alle *erreichbaren* Markierungen eines Netzes bezieht. Es gibt jedoch hinreichende Bedingungen für Sicherheit, die auf strukturellen Eigenschaften von S/T-Netzen gründen [Sta90]. Ähnliche Eigenschaften wie Sicherheit in S/T-Netzen lassen sich auch für andere Petrinetz-Typen definieren.

2.2.3 Klassifikation von Petrinetz-Typen

Die lange Entwicklung auf dem Gebiet der Petrinetze brachte eine Vielzahl von Petrinetz-Typen hervor. Jeder dieser Petrinetz-Typen hat seine unbestrittene theoretische oder praktische Bedeutung. Es fehlte nicht an Versuchen, diese Vielzahl von Petrinetz-Typen zu klassifizieren bzw. mit Hilfe von Parametern eines allgemeinen Netztyps zu beschreiben.

Allgemeine Petrinetz-Theorie

Zunächst waren die wesentlichen Beiträge zur Petrinetz-Theorie überschaubar, so dass sie unter dem Stichwort *Allgemeine Petrinetz-Theorie* zusammengefasst wurden. Carl Adam Petri begründet die Notwendigkeit einer solchen Allgemeinen Petri-

netz-Theorie [Pet77b, Pet80], um Aussagen zu einer Klasse von Petrinetzen zu treffen. In einem Sammelband [Bra80] finden sich neben S/T-Netzen bereits weitere Petrinetz-Typen wie Zeit-Petrinetze [Sif80] oder Bedingungs-Ereignis-Netze (B/E-Netze) [GLT80]. Begriffe und Notationen von Petrinetzen werden für Petrinetz-Graphen, Kausalnetze, B/E-Netze und S/T-Netze kompakt dargestellt [GS80, BF86].

Später finden sich überblicksartige Darstellungen von Petrinetz-Typen wie Elementare Netzsysteme [Thi87, Roz87], S/T-Netze [Rei87], Prädikat/Transitions-Netze (Pr/T-Netz) [Gen87], *Coloured Petri Nets* [Jen87], FIFO-Netze [Rou87] und Stochastische Netze [Pag87] in einem Sammelband [BRR87]. Auch in einem weiteren Sammelband [RR98] finden sich Darstellungen von grundlegenden Petrinetz-Typen (Elementare Netzsysteme [RE98], S/T-Netze [DR98]) sowie Prinzipien von *high level* Netzen [Smi98] und Stochastischen Netzen (Zeit-Petrinetze subsumierend) [AMBD98].

Eine weitere Quelle für einen Überblick über Petrinetz-Typen sind die verschiedenen Werkzeug-Kataloge, die im Laufe der Zeit zusammengestellt wurden. Sie haben den Vorteil, für einen bestimmten Zeitpunkt relevante Petrinetz-Werkzeuge mit den von ihnen bearbeiteten Petrinetz-Typen zusammenzuführen. Werkzeugkataloge reichen von einer Aufzählung von Leistungsmerkmalen und Referenzen (zuletzt [Fel93]) über Tabellen [Gar98] und Web-basierte Datenbanken mit Suchfunktion [PNT00] bis hin zu kritischen Evaluationen [Stö97]. Insbesondere die Web-basierte Datenbank von Petrinetz-Werkzeugen [PNT00] spielt eine große Rolle, wenn Anwender Werkzeuge für Petrinetze eines bestimmten Petrinetz-Typs suchen.

Kantentypen

Ein wichtiger Schritt zur Klassifikation von Petrinetz-Typen waren spezielle Klassifikationen. Eine Klassifikation von Charles Lakos und Søren Christensen beschäftigt sich mit Kantentypen in erweiterten *Coloured Petri Nets* [LC94]. Das Ergebnis war, dass für erweiterte *Coloured Petri Nets* 4 Kantentypen ausreichen, um jeden bis dahin beschriebenen Kantentyp zu emulieren. Dabei werden komplementäre Stellen verwendet und die Bedeutung und Syntax von Kanteninschriften in *Coloured Petri Nets* ausgenutzt. Wir werden diesen Ansatz in Abschn. 3.4 aufgreifen.

Zeit-Petrinetze

Eine Klassifikation von Zeit-Petrinetzen wurde von Peter Starke erarbeitet [Sta95]. Zeit-Petrinetze wurden hinsichtlich ihrer Zeitanschriften, ihrer Uhren und ihrer Schaltstrategien untersucht. Uhren sind entweder an Marken oder Transitionen gebunden. Zeitanschriften geben eine Dauer oder ein Intervall an. Die Zeitanschrift bedeutet bezogen auf die Uhrenstellung, dass Marken gültig (und damit „schaltbar“) oder ungültig sind bzw. dass eine Transition schalten kann oder nicht. Die Schaltstrategie

bestimmt, ob eine Transition keinen Schaltzwang hat, so früh wie möglich schalten muss oder so spät wie möglich schalten darf.

Parametrisierte Petrinetz-Typen

Seit Mitte der 1990er Jahre wird verschiedentlich versucht, Petrinetz-Typen allgemein zu klassifizieren. Ein erster Versuch war, ein allgemeines Petrinetz zu beschreiben [Bra92]. Damit gehen jedoch die Eigenschaften spezieller Petrinetz-Typen und deren Analysemöglichkeiten verloren [Sta90].

Der Ansatz parametrisierter Petrinetz-Typen überwindet diese Schwierigkeiten, da je nach Wahl der Parameter Eigenschaften für Netze des entsprechenden Typs gelten. Zu Ansätzen parametrisierter Petrinetz-Typen zählen wir *Abstrakte Petrinetze* [Pad96] (darauf aufbauend algebraisch allgemeine Petrinetze [Juh99]) und den *Petrinetz-Würfel* [KW98]. Diese Ansätze basieren auf der Beobachtung, dass sich für viele Petrinetz-Typen die Markierung einer Stelle wie ein Monoid verhält [MM90]. Abstrakte Petrinetze unterscheiden *low level* und *high level* Abstrakte Petrinetze. *Low level* Abstrakte Petrinetze definieren für verschiedene *low level* Petrinetz-Typen verschiedene Algebren (Monoide[1]) über den Elementen der Stellenmenge. *High level* Abstrakte Petrinetze definieren die Algebra über dem Datentyp des entsprechenden Netzes.

Der Petrinetz-Würfel definiert die Algebra der Markierung als freie Konstruktion (freies Monoid) [EMC+99] über der Menge von Marken. Diese Menge von Marken ist einelementig (*low level* Petrinetz-Typen) oder beliebig (*high level* Petrinetz-Typen).

Beide Ansätze setzen mit der algebraischen Beschreibung einer Markierung die Atomarität von Marken zunächst nicht voraus. Im Petrinetz-Würfel wird durch eine entsprechende Definition der Schaltregel gesichert, dass die Atomarität von Marken nicht verletzt wird. In der vorliegenden Arbeit wird dagegen von der Atomarität der Marken ausgegangen. Markierungen von Stellen werden entsprechend aus atomaren Marken zusammengesetzt. Damit sind beispielsweise XML-Netze [LO01] kein Petrinetz-Typ im Sinne dieser Arbeit, da Marken in diesen Netzen nicht unteilbar sind.

Die beiden Ansätze Abstrakte Petrinetze und Petrinetz-Würfel formulieren die *klassische* Schaltregel für Petrinetze. Die klassische Schaltregel schaltet genau eine Transition in einem Schritt. In Abstrakten Petrinetzen wird die Schaltregel separat für jede Instanz (also jeden Petrinetz-Typ) formuliert. Im Petrinetz-Würfel ist diese Schaltregel inhärent, das heißt, für jeden Petrinetz-Typ, der mit dem Petrinetz-Würfel formuliert werden kann, ergibt sich die klassische Schaltregel, ohne sie explizit für die Parameter des Petrinetz-Typs anzupassen. Für den Petrinetz-Würfel wurde ein Werkzeug implementiert [Web99], mit dem Petrinetze als Instanzen eines parametrisierten Petrinetz-Typs mit 2 Parametern (Menge der Marken, Algebra der Markierung einer Stelle) ausgedrückt werden und für die das Markenspiel simuliert wird.

[1] Für den Invarianten-Kalkül werden auch Gruppen verwendet.

Petrinetz-Hyperwürfel

In der vorliegenden Arbeit werden wir den Ansatz des Petrinetz-Würfels einer inhärenten Schaltregel für den *Petrinetz-Hyperwürfel* weiter verfolgen und eine parametrisierte Schaltregel einführen. Diese parametrisierte Schaltregel enthält die klassische Schaltregel als eine Instanz unter anderen. Für andere Schaltregeln (z. B. die Schrittschaltregel) ist es nötig, besondere Anforderungen (z. B. Konflikte zwischen Transitionen) zu beachten. Die parametrisierte Schaltregel formuliert solche Anforderungen mit Hilfe von Parametern. In diesem Sinne ist die inhärente, parametrisierte Schaltregel des in der vorliegenden Arbeit eingeführten Petrinetz-Hyperwürfels allgemeiner als bisherige Ansätze.

2.3 Petrinetz-Werkzeuge

In diesem Abschnitt werden Petrinetz-Werkzeuge unter verschiedenen Gesichtspunkten betrachtet. Ein *Petrinetz-Werkzeug* nennen wir jede Software, mit deren Hilfe ein Petrinetz bearbeitet, dauerhaft gesichert, simuliert (ausgeführt) oder analysiert werden kann. In Abschn. 2.3.1 werden Klassen von Petrinetz-Werkzeugen vorgestellt. Abschnitt 2.3.2 stellt verschiedene Petrinetz-Werkzeuge unter dem Blickwinkel ihrer Kooperation dar. Und in Abschn. 2.3.3 werden Dateiformate von Petrinetz-Werkzeugen diskutiert.

2.3.1 Werkzeugklassen

Es gibt zahlreiche Petrinetz-Werkzeuge für unterschiedliche Anforderungen. Dem grafischen Charakter von Petrinetzen angemessen gibt es sehr viele grafische Werkzeuge zum Modellieren, Simulieren und Analysieren von Petrinetzen (z. B. Design/CPN [CPN00a], Renew [Ren99], PEP [Gra97]). Es gibt jedoch auch Werkzeuge, die auf eine grafische Benutzerschnittstelle verzichten und den mathematischen Aspekt von Petrinetzen besonders betonen (z. B. INA [RS98], Maria [Mar98] und LoLA [Schoo]). Werkzeuge der ersten Art richten sich vor allem an Modellierer und Anwender von Petrinetzen. Wohingegen Werkzeuge der zweiten Art vor allem für Petrinetz-Experten gedacht sind. Durch den Verzicht auf grafische Programmelemente erreichen solche Werkzeuge einen beachtlichen Umfang implementierter Analysealgorithmen. Ein besonderes Augenmerk liegt dabei auf der Effizienz der implementierten Algorithmen. Verschiedentlich werden solche Werkzeuge in grafische Petrinetz-Werkzeuge integriert (z. B. [KO98], [Haa98]). Damit lassen sich die Vorteile beider Welten kombinieren.

Ein Petrinetz-Werkzeug besteht im Allgemeinen aus verschiedenen Komponenten, die wir im Folgenden nach ihrer Funktion betrachten. Jedes Petrinetz-Werkzeug benötigt eine Komponente zur Erfassung eines Petrinetzes aus einer Datei oder im interaktiven Dialog mit einem Nutzer. Viele Petrinetz-Werkzeuge besitzen einen Mar-

kenspielsimulator, der es einem Nutzer gestattet, die Schaltregel des entsprechenden Petrinetz-Typs auf seine Netze anzuwenden. Dem Modellierer eines Petrinetzes wird somit ein Gefühl für das modellierte Netz vermittelt.

Petrinetz-Werkzeuge, die für Performanzanalyse eingesetzt werden, benötigen Analysekomponenten für Simulationsläufe. Hier kommt es vor allem darauf an, Analyseergebnisse bezogen auf das Anwendungsgebiet des Werkzeuges zu interpretieren und Aussagen über das mit dem entsprechenden Petrinetz modellierte System zu treffen. Eine weitere Klasse von Petrinetz-Werkzeugkomponenten wird als Laufzeitsystem eingesetzt, z. B. im Bereich Workflow das Werkzeug COSA [COS01].

Viele Petrinetz-Werkzeuge insbesondere im akademischen Umfeld enthalten Analysekomponenten z. B. zur Berechnung von Eigenschaften eines Petrinetzes, wie die *extended free choice* Eigenschaft, oder zur Berechnung von Invarianten eines Petrinetzes. Häufig implementieren solche Analysekomponenten Ergebnisse der Petrinetz-Theorie. Für den Nachweis von Systemeigenschaften binden einige Petrinetz-Werkzeuge externe *model checking* Werkzeuge an. Andere implementieren selbst ein *model checking* Werkzeug (z. B. LoLA [Schoo]).

Schließlich werden Petrinetze in einigen Werkzeugen transformiert. Das Ziel ist dabei, ein Petrinetz eines bestimmten Typs zu Analysezwecken in ein Petrinetz eines anderen Typs zu übersetzen, der vorzugsweise umfassend mit Petrinetz-theoretisch fundierten Methoden untersucht werden kann. Meist wird in S/T-Netze transformiert [Schoo]. Eine andere Möglichkeit ist die Übersetzung eines Petrinetzes in eine Programmiersprache (z. B. eine parallele im Werkzeug PEP [PEP98]) zur Ausführung als Programm.

2.3.2 Metawerkzeuge

In diesem Abschnitt sollen Petrinetz-Werkzeuge und Werkzeuge aus verwandten Gebieten unter dem Aspekt der Kooperation betrachtet werden. Zu unterscheiden sind verschiedene Formen der Kooperation – Integration, Kapselung und Transformation. Petrinetz-Werkzeuge, die vor allem als Entwicklungswerkzeuge eingesetzt werden (wie PEP [PEP98] und Design/CPN [CPNooa]) integrieren Werkzeugkomponenten unter einer Plattform. Einen etwas anderen Ansatz verfolgt das Werkzeug CPN-Ami [CPNoob], das *eigenständige* Petrinetz-Werkzeuge (auch solche mit einer eigenen grafischen Schnittstelle) mit einer einheitlichen Benutzerschnittstelle kapselt. Vergleichbare Ansätze, wenn auch weniger integrativ, findet man auch für andere formale Methoden. So wird versucht, verschiedene Spezifikationssprachen und Verifikationswerkzeuge ineinander zu überführen, um die jeweiligen Vorteile der formalen Methoden zu nutzen [KG99, SMB97]. Ein anderer Ansatz [MCK99] nutzt die jeweilige Stärke verschiedener *model checking* Werkzeuge zur Überprüfung verschiedener Eigenschaften in Transitionssystemen.

Die oben beschriebenen Ansätze richten sich an verschiedene Anwendergruppen.

Die Entwicklungswerkzeuge PEP und Design/CPN sind gedacht zur Modellierung und Analyse verteilter bzw. paralleler Systeme. Auch CPN-Ami wendet sich an diesen Anwenderkreis. Während die transformierenden Werkzeuge aus dem Umfeld der Formalen Methoden einen Ausweis ihrer Leistungsfähigkeit darstellen.

2.3.3 Dateiformate

Zur dauerhaften Speicherung von Petrinetzen benötigen Petrinetz-Werkzeuge ein Verfahren, um hergestellte Petrinetze zu speichern und zu reproduzieren. Zunächst hat jedes Petrinetz-Werkzeug ein eigenes Dateiformat. Dies kann ein *ad hoc* entwickeltes Format oder ein motiviertes Format (z. B. [BKK95]) sein.

Für die Kooperation verschiedener Petrinetz-Werkzeuge bzw. Werkzeuge, die Petrinetze als Eingabesprache verwenden, muss ein Format verwendet werden, auf dass sich wenigstens die beteiligten Partner einigen können. Weil ein solches Dateiformat für die Kooperation zwischen Petrinetz-Werkzeugen (also auch für die Außendarstellung der Petrinetz-Welt) wichtig ist, fehlt es nicht an Versuchen, ein einheitliches Dateiformat für Petrinetze zu entwickeln, mit dem sich möglichst alle Petrinetz-Werkzeuge verbinden lassen. Ein einheitliches Dateiformat kann sich auf zwei Arten durchsetzen. Zum einen kann ein Werkzeug durch Marktmacht einen Quasistandard für einen bestimmten Bereich definieren. Zum anderen, vor allem in heterogenen Bereichen, kann ein Gremium einen Standard entwickeln. Petrinetze und ihre Werkzeuge sind ein Beispiel für solch einen heterogenen Bereich. Außerdem gibt es kein Werkzeug, das für alle Petrinetz-Typen verwendet werden kann und alle Petrinetze in seinem Dateiformat widerspiegelt. Mit der Entwicklung eines internationalen Standards für *high level* Petrinetze begann auch die Diskussion (siehe [BBK+00]) um ein allgemeines Dateiformat für Petrinetze. Ausgangspunkt aller Beiträge zu diesem Thema ist die naheliegende Beobachtung, dass alle Petrinetze aus Stellen, Transitionen und Kanten sowie weiteren Elementen bestehen.

Ein erster Vorschlag für ein Standardformat für Petrinetze war die *Abstract Petri Net Notation* [BKK95] eine an LaTeX angelehnte *Markup*-Sprache. Mit der Entwicklung von XML ergaben sich für diesen Ansatz des *Markup* neue Perspektiven, da für die meisten Programmiersprachen Parser und Objektmodelle für XML zur Verfügung stehen. Auf der Petrinetz-Konferenz 2000 wurde eine Standardisierung für Petrinetz-Dateiformate eingeleitet [BBK+00]. Inzwischen gibt es einige Petrinetz-Werkzeuge, die auf XML basierte Dateiformate setzen (z. B. [Ren99, LM98]).

3 Petrinetz-Hyperwürfel

Der Petrinetz-Würfel [KW98] wurde entwickelt, um Typen von Petrinetzen zu klassifizieren. In [KW98] gelang dies für klassische Petrinetz-Typen. Eine Erweiterung auf andere Petrinetz-Typen wurde lediglich skizziert.

Klassische Petrinetz-Typen sind solche, deren Schaltregel nur von den Marken auf den Stellen im Vor- bzw. Nachbereich *einer* schaltenden Transition abhängt. Marken im Vorbereich der Transition werden von der jeweiligen Stelle entfernt, und es werden Marken zu Stellen im Nachbereich der Transition hinzugefügt. Das impliziert, dass klassische Petrinetz-Typen nur gewöhnliche Kanten in Petrinetzen zulassen. Eine solche Kante ist eine in eine Transition eingehende Kante (von einer Stelle im Vorbereich der Transition) oder eine aus der Transition ausgehende Kante (zu einer Stelle im Nachbereich der Transition). *Nicht klassische* Petrinetze sind dann solche, die diesen Prozess des Schaltens verändern, indem weitere Operationen ausgeführt werden (z. B. das Zurücksetzen von Uhren in Zeit-Petrinetzen), mehrere Transitionen gemeinsam schalten (z. B. Schaltregel des maximalen Schrittes), die Auswahl der zu schaltenden Transitionen gesteuert wird (z. B. Priorität), ein komplexes Schaltverhalten angenommen wird (z. B. in Signal-Ereignis-Netzen) oder weitere Kanten zwischen Stellen und Transitionen mit besonderer Bedeutung existieren.

In diesem Kapitel wird der Petrinetz-Würfel zu einem Petrinetz-Hyperwürfel erweitert, so dass auch nicht klassische Petrinetz-Typen beschrieben werden können. Ein derartiges Konzept der Zusammenfassung aller Petrinetz-Typen reduziert die Unterschiede zwischen Petrinetzen verschiedener Typen auf das Wesentliche. In unserer Betrachtungsweise besteht ein Petrinetz aus syntaktischen Objekten, über und mit denen die Schaltregel operiert. Analysemethoden für Petrinetze verschiedener Typen können so *begründet* übertragen bzw. nicht übertragen werden. Beispielsweise ist der Kalkül für Stellen-Invarianten in Signal-Netzsystemen gleich dem für S/T-Netze mit Testkanten. Andererseits unterscheiden sich im Allgemeinen die Erreichbarkeitsgraphen für Netze mit und ohne Schrittschaltregel.

Die Dimensionen sowohl des Petrinetz-Würfels als auch des Petrinetz-Hyperwürfels sind orthogonal zueinander. Das heißt, eine beliebige Kombination eines Wertes für jede Dimension erzeugt einen Petrinetz-Typ.

Dieses Kapitel ist wie folgt strukturiert. Zunächst führen wir den Begriff des Petrinetz-Graphen ein (Abschn. 3.1). Anschließend stellen wir in Abschn. 3.2 die Basisdimensionen des Petrinetz-Hyperwürfels vor, mit deren Hilfe bereits die „klassische“ Schaltregel beschrieben werden kann. Abschnitt 3.3 führt den Begriff des Labels ein.

Dieser Begriff ist die Grundlage für die Petrinetz-Beschreibungssprache PNML und die Basisversion des Petrinetz-Kerns. In Abschn. 3.4 klassifizieren wir Kantentypen – eine spezielle Form von Labeln. Abschnitt 3.5 stellt die parametrisierte Schaltregel des Petrinetz-Hyperwürfels vor, mit der alle Schaltregeln von Petrinetzen beschrieben werden können. Und schließlich definieren wir in Abschn. 3.6 Petrinetz-Typen und Petrinetze formal. Außerdem betrachten wir dort einige Beispiele von Petrinetz-Typen aus der Literatur als Instanzen des Petrinetz-Hyperwürfels.

3.1 Petrinetz-Graph

In der Literatur werden Petrinetze meist als Graphen mit zwei verschiedenen Knotenmengen und einer Relation zwischen ihnen definiert.

Definition 3.1.1 (Netz (trad.)). *Ein* Netz $N = (P, T; F)$ *besteht aus zwei disjunkten Mengen P und T sowie einer Relation $F \subseteq (P \times T) \cup (T \times P)$. Die Elemente von P heißen* Stellen*, die Elemente von T* Transitionen *und die Elemente von F* Kanten*.* *

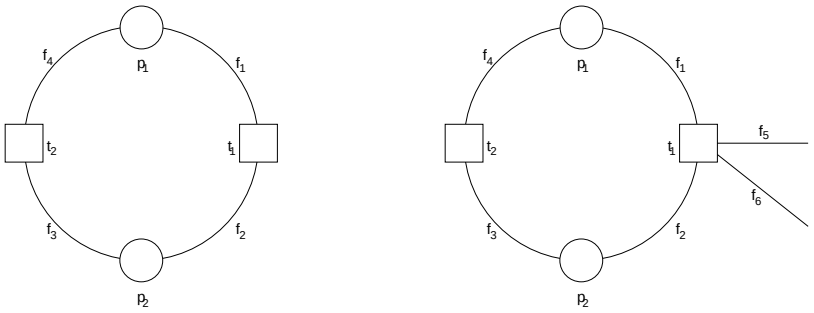
Definition 3.1.1 ist jedoch für unsere Zwecke weniger geeignet, da eine Stelle und eine Transition maximal mit je einer eingehenden und einer ausgehenden Kante verbunden sind. Eine Multimenge von Relationen ließe zwar mehrere Kanten (gleicher Richtung) zwischen einer Stelle und einer Transition zu. Die Betrachtung genau einer Kante ist damit aber schwierig. Außerdem lässt sich ein Typkonzept für Kanten mit obigem Ansatz nicht allgemein erfassen, wie wir in Abschn. 3.4 sehen werden. Genausowenig können mit diesem Ansatz, die Petrinetze einiger Petrinetz-Werkzeuge (wie [HVA97, PNKoo]), die mehrere gleichgerichtete Kanten zwischen Stellen und Transitionen zulassen, adäquat dargestellt werden.

Wir definieren deshalb das Netz (als Bestandteil jedes Petrinetzes) ein wenig anders (ähnlich für *Coloured Petri Nets* [Jen92]), behalten aber die eingeführten Bezeichnungen bei.

Definition 3.1.2 (Netz). *Ein* Netz $N = (P, T, F; R)$ *besteht aus drei paarweise disjunkten endlichen Mengen P (*Stellen*), T (*Transitionen*) und F (*Kanten*) sowie einer Abbildung (*Kantenrelation *genannt) $R : F \to P \times T$. Jede Kante verbindet eine Stelle mit einer Transition des Netzes. Es gilt damit für alle $f \in F$, dass es ein $p \in P$ und ein $t \in T$ so gibt, dass $R(f) = (p, t)$* *

Für die Komponenten P, T, F und R eines Netzes $N = (P, T, F; R)$ schreiben wir auch P_N, T_N, F_N und R_N. Ein Element der Menge $P_N \cup T_N$ heißt auch *Knoten*, und ein Element der Menge $P_N \cup T_N \cup F_N$ heißt auch *Netzelement*.

Mit der Def. 3.1.2 haben Kanten nun keine implizite Semantik, die mit der Richtung wie in Netzen nach Def. 3.1.1 angedeutet wird. Die Semantik der Kanten wird hier



(a) Netz N_1

(b) Kein Netz

(c) Kein Netz, da $R(f)$ nicht definiert

Abbildung 3.1: Beispiel/Gegenbeispiel Netz

dagegen explizit mit einem Kantentyp festgelegt (siehe Abschn. 3.4). Die Menge der Kantentypen für ein Petrinetz wird von seinem Petrinetz-Typ festgelegt (siehe Abschn. 3.6).

Grafisch stellen wir eine Stelle durch eine Ellipse (Kreis), eine Transition durch ein Rechteck (Quadrat) und eine Kante f der Kantenrelation $R(f) = (p, t)$ entsprechend durch eine Verbindungslinie zwischen der Stelle p und der Transition t dar. Eine Kante erhält zunächst keine Pfeilspitze wie in Petrinetzen sonst üblich, da die Pfeilspitze einer Kante die graphische Repräsentation des Kantentyps (siehe Abschn. 3.4) ist.

Abbildung 3.1(a) zeigt ein Beispiel für ein Netz nach Def. 3.1.2. Kanten erhalten (nur) hier einen Bezeichner, um den Zusammenhang zur folgenden mathematischen Repräsentation herzustellen.

$$N_1 = (\{p_1, p_2\}, \{t_1, t_2\}, \{f_1, f_2, f_3, f_4\}; \\ \{(p_1, f_1, t_1), (p_2, f_2, t_1), (p_2, f_3, t_2), (p_1, f_4, t_2)\}) \quad (3.1)$$

Gleichung (3.1) ist die mathematische Repräsentation des Netzes aus Abb. 3.1(a). Die Abb. 3.1(b) stellt dagegen kein Netz im Sinne der Def. 3.1.2 dar, da für die Kanten f_5 und f_6 die Kantenrelation R (entgegen der Def. 3.1.2) nicht definiert ist. Auch die Ansammlung von Elementen in Abb. 3.1(c) ist kein Netz.

Mit der Def. 3.1.2 können wir nicht mehr den Vor- bzw. Nachbereich einer Transition (das ist die Menge der Vor- bzw. Nachplätze) definieren, wie es in der klassischen

Petrinetz-Theorie (z. B. [Rei86]) für Petrinetz-Graphen üblich ist, da die „Kantenrichtung“ vom Typ der Kante gegeben wird. Erst für Petrinetze eines bestimmten Typs kann in Abhängigkeit von den Kantentypen der Vor- und Nachbereich von Transitionen definiert werden. Für einige Kantentypen wird es sogar schwierig werden, eine Kantenrichtung festzulegen (vgl. Abschn. 3.4). Wir können jedoch mit der Def. 3.1.2 die Umgebung einer Transition (bzw. einer Stelle) definieren.

Definition 3.1.3 (Umgebung einer Transition). *Sei N ein Netz. Für $t \in T_N$ heißt die Menge $\overset{\circ}{t} = \{p \mid$ es gibt ein $f \in F_N$ so, dass $R_N(f) = (p,t)\}$ die* Umgebung der Transition t. *

Die Umgebung einer Transition sind Stellen.

Häufig benötigen wir jedoch das Umfeld einer Transition. Und das ist die Umgebung der Transition einschließlich der Kanten zwischen Stellen der Umgebung und der Transition

Definition 3.1.4 (Umfeld einer Transition). *Sei N ein Netz. Für $t \in T_N$ heißt die Menge $\overset{\bullet}{t} = \overset{\circ}{t} \cup \{f \mid$ es gibt ein $p \in P_N$ so, dass $R_N(f) = (p,t)\}$ das* Umfeld der Transition t. *

Zu einem Knoten eines Netzes können wir alle *adjazenten* Kanten bestimmen. Das sind alle die Kanten, die diesen Knoten mit einem anderen Knoten des Netzes verbinden.

Definition 3.1.5 (Menge der adjazenten Kanten). *Sei N ein Netz und $k \in P_N \cup T_N$ ein Knoten des Netzes. Dann ist die* Menge der adjazenten Kanten *von k die Menge $\overset{\triangle}{k} = \{f \mid$ es gibt ein $k' \in P_N \cup T_N$ so, dass $R_N(f) = (k',k)$ oder $R_N(f) = (k,k')\}$.* *

Schließlich bestimmen wir die Stelle einer Kante bzw. die Transition einer Kante. Die Transition, zu der eine Kante Element der Menge der adjazenten Kanten ist, heißt *Transition der Kante*. Die Stelle, zu der eine Kante adjazent ist, heißt *Stelle der Kante*. Die Transition einer Kante f bezeichnen wir mit f_{T} , und die Stelle der Kante bezeichnen wir mit f_{P}. ,

3.2 Basisdimensionen

In diesem Abschnitt werden die Basisdimensionen des Petrinetz-Hyperwürfels beschrieben, wie sie bereits im Petrinetz-Würfel [KW98] verwendet wurden. Jeder Petrinetz-Typ oder besser jede Instanz eines solchen Typs (also jedes Petrinetz) legt die genaue Ausprägung der Basisdimensionen fest. Mit den Basisdimensionen können, wie mit dem Petrinetz-Würfel [KW98] gezeigt, verschiedene Versionen klassischer Petrinetze definiert werden.

Die Werte der Basisdimensionen legen fest, welche Marken in Netzen des jeweiligen Typs verwendet werden (Markenmenge) und wie Marken zu Markierungen von Stellen zusammengesetzt werden (Markierungsstruktur). Beispielsweise ist eine beliebige mehrelementige Markenmenge kombiniert mit einer Multimenge als Markierungsstruktur dem Petrinetz-Typ *Coloured Petri Nets* [Jen92] vergleichbar.

In den folgenden Abschnitten beschreiben wir die Markenmenge (Abschn. 3.2.1) und die Markierungsstruktur (Abschn. 3.2.2). Sowohl die Markierung eines Netzes als auch die Inschrift einer Kante wird mit diesen Dimensionen formuliert. Abschnitt 3.2.3 führt die Schaltregel für klassische Petrinetz-Typen ein. Diese Schaltregel ist damit nur abhängig von den beiden Werten für die ersten beiden Dimensionen. Wir haben für den Petrinetz-Würfel [KW98] noch eine dritte Dimension – die Kantenflussstruktur – eingeführt. Diese Dimension schränkt die Menge der zulässigen Kanteninschriften ein, um ungewöhnliche Kanteninschriften (z. B. 0 für S/T-Netze) auszuschließen oder um feine Unterschiede in der Definition verschiedener *high level* Netze (z. B. Algebraische Netze [Krä85, Bil89, Rei91b] im Unterschied zu *Coloured Petri Nets* [Jen92]) zu beschreiben. Da unser Augenmerk hier auf der Schaltregel von Petrinetzen liegt, ist uns diese Dimension nicht wichtig. Wir werden später lediglich festlegen, dass nur Kanteninschriften beachtet werden, die in Elemente der Markierungsstruktur ausgewertet werden können.

3.2.1 Markenmenge

Die erste Dimension des Petrinetz-Hyperwürfels nennen wir *Markenmenge*. Mit ihr wird die Menge aller unterscheidbaren Token eines Netzes zusammengefasst und mit A bezeichnet.

Der Wert der Markenmenge (also die Menge A) legt bereits fest, ob der Petrinetz-Typ *low level* Petrinetze oder *high level* Petrinetze definiert. Für *low level* Petrinetze ist $A = \{\bullet\}$ und für *high level* Petrinetze ist A beliebig. In *low level* Petrinetzen haben also alle Marken den gleichen Wert (*black token*, schwarze Marke), während Marken in *high level* Petrinetzen voneinander verschiedene Werte (*colour*, Farben) haben können. Von der Datentypspezifikation einiger *high level* Petrinetze abstrahieren wir hier; wir betrachten alle Marken als Elemente der Menge A.

3.2.2 Markierungsstruktur

Die zweite Dimension des Petrinetz-Hyperwürfels ist die *Markierungsstruktur*. Sie legt fest, wie die Marken auf den Stellen und Terme über Marken in Anschriften an Netzelementen zu Markierungen (auf Stellen) bzw. Ausdrücken über Markierungen (in Kanteninschriften und Aktivierungsbedingungen von Transitionen) zusammengesetzt werden. Die Markierungsstruktur ist damit einem Container für Marken vergleichbar. Dazu bedienen wir uns der in Abschn. 2.1.5 eingeführten Klasse von Markierungsal-

gebren. Die Markierungsstruktur angewendet auf eine Markenmenge A ist dann eine Markierungsalgebra $\mathfrak{A}(A) = (B, \langle \varepsilon, \oplus, \ominus \rangle)$ mit $u : A \to B$ und für alle $a \in A$ sei $u(a) \in B$. Es gelten die Eigenschaften (2.11) und (2.12); die Elemente aus B, die mit u aus A hervorgehen, sind also atomar.

Die Markierungsstruktur angewendet auf eine Markenmenge ist die Algebra aus einer Klasse von Markierungsalgebren, die typisch [EMC+99, KW98] für eine gegebene Menge (hier die Markenmenge) ist und die die Atomarität dieser Menge respektiert. Jede Markierungsalgebra, deren Klasse für den Petrinetz-Hyperwürfel verwendet wird, hat die Eigenschaften, die wir in Abschn. 2.1.5 eingeführt haben. Zusätzlich definiert jede verwendete Klasse weitere Eigenschaften für ihre Algebren, die Basismenge B der jeweiligen Algebra sowie die Abbildung u.

In dieser Arbeit werden in den folgenden Unterabschnitten einige konkrete Markierungsstrukturen als Teilklassen der Klasse von Markierungsalgebren aus Abschn. 2.1.5 eingeführt. Weitere konkrete Markierungsstrukturen können nach dem selben Schema definiert werden.

Markierungen als Multimenge

Die meisten Definitionen von Petrinetzen (z. B. S/T-Netze [Rei87] oder *Coloured Petri Nets* [Jen92]) lassen mehrere gleiche Marken auf einer Stelle zu. Die Markierung einer solchen Stelle ist also eine Multimenge (vgl. Abschn. 2.1.3). Wir definieren $\mathfrak{M}\mathfrak{s} \subseteq \mathfrak{A}$ als eine Algebra $\mathfrak{M}\mathfrak{s}(A) = (B, \langle \varepsilon, \oplus, \ominus \rangle)$ mit

$$B = \mathcal{B}(A) \tag{3.2a}$$

$$\forall a \in A : u(a) = [a] \tag{3.2b}$$

Die Basismenge B dieser Algebra ist damit die Menge aller Multimengen über A (jedes Element der Basismenge ist mithin eine Multimenge über A). Es gelten folgende Eigenschaften für $\oplus$ und $\ominus$ mit $x, y \in B$:

$$x \oplus y = y \oplus x \tag{3.3a}$$

$$(x \oplus y) \ominus y = x \tag{3.3b}$$

Danach ist $\oplus$ kommutativ, und ein Element, das hinzugefügt wird, kann auch wieder entnommen werden. Außerdem definieren wir für $\varepsilon \in B$, $\oplus$ und $\ominus$:

$$\varepsilon = [] \tag{3.4a}$$

$$x \oplus y = x \sqcup y \tag{3.4b}$$

$$x \ominus y = \begin{cases} x \boxminus y & \text{wenn } y \sqsubseteq x \\ \text{nicht definiert} & \text{sonst} \end{cases} \tag{3.4c}$$

Gleichung (3.4a) definiert die leere Multimenge als das neutrale Element dieser Algebra. Gleichung (3.4b) bestimmt, dass die Vereinigung zweier Multimengen Element der Markierungsstruktur ist. Und Gl. (3.4c) sagt aus, dass die Operation $\ominus$ nur definiert ist, wenn die Multimenge y Teilmultimenge der Multimenge x ist.

Markierungen als Menge

Elementare Netzsysteme [Thi87] und Prädikat-/Transitions-Netze (Pr/T-Netze) [Gen87] definieren, dass jede Marke höchstens einmal auf einer Stelle vorkommen darf. Die Markierung einer solchen Stelle ist also eine Menge von Marken (vgl. Abschn. 2.1.1). Wir definieren $\mathfrak{Mg} \subseteq \mathfrak{A}$ als eine Algebra $\mathfrak{Mg}(A) = (B, \langle \varepsilon, \oplus, \ominus \rangle)$ mit

$$B = \mathcal{P}(A) \tag{3.5a}$$

$$\forall a \in A : u(a) = \{a\} \tag{3.5b}$$

Die Basismenge B der Algebra ist damit die Menge aller Teilmengen über A. Die Operation $\oplus$ ist kommutativ, mit $x, y \in B$ gilt:

$$x \oplus y = y \oplus x \tag{3.6}$$

Außerdem definieren wir für $\varepsilon \in B$, $\oplus$ und $\ominus$:

$$\varepsilon = \{\} \tag{3.7a}$$

$$x \oplus y = \begin{cases} x \cup y & \text{wenn } x \cap y = \{\} \\ \text{nicht definiert} & \text{sonst} \end{cases} \tag{3.7b}$$

$$x \ominus y = \begin{cases} x \setminus y & \text{wenn } y \subseteq x \\ \text{nicht definiert} & \text{sonst} \end{cases} \tag{3.7c}$$

Gleichung (3.7a) definiert die leere Menge als neutrales Element dieser Markierungsstruktur. Und die Gleichungen (3.7b) und (3.7c) besagen, dass ein Element der konstituierenden Menge A nicht verschwindet.

Markierungen als Sequenz

Es gibt einige wenige Petrinetz-Typen (z. B. FIFO-Netze [Rou87]), die es zulassen, die Markierung einer Stelle als geordnete Liste (z. B. *first in first out* Kanal) aufzufassen. Eine Marke wird an dem einen Ende der Liste hinzugefügt und am anderen entnommen. Mathematisch können wir eine derartige Struktur als Sequenz betrachten (vgl. Abschn. 2.1.4).

Wir definieren $\mathfrak{Sq} \subseteq \mathfrak{A}$ als eine Algebra $\mathfrak{Sq}(A) = (B, \langle \varepsilon, \oplus, \ominus \rangle)$ mit

$$B = A^* \tag{3.8a}$$

$$\forall a \in A : u(a) = (a) \tag{3.8b}$$

Die Basismenge B dieser Algebra ist damit die Menge aller endlichen Sequenzen über der Menge A. Es gilt für die Operationen $\oplus$ und $\ominus$ mit $x, y \in B$

$$(x \oplus y) \ominus y = x \tag{3.9}$$

Das heißt eine Teilsequenz y kann von einer Sequenz $z = x \oplus y$ nur entfernt werden, wenn y Suffix von z ist. Für $\varepsilon \in B$, $\oplus$ und $\ominus$ definieren wir:

$$\varepsilon = () \tag{3.10a}$$

$$x \oplus y = x \circ y \tag{3.10b}$$

$$x \ominus y = \begin{cases} z & \text{wenn es ein } z \in B \text{ so gibt, dass } z \circ y = x \\ \text{nicht definiert} & \text{sonst} \end{cases} \tag{3.10c}$$

Gleichung (3.10a) bestimmt die leere Sequenz zum neutralen Element der Markierungsstruktur. Gleichung (3.10b) besagt, dass jede zusammengesetzte Sequenz ein Element der Markierungsstruktur ist, und Gl. (3.10c) definiert, dass die Operation $\ominus$ nur zulässig ist, wenn y Suffix von x ist.

Markierungen als Stapel

Neben den in den vorangegangenen Abschnitten definierten Markierungsstrukturen sind weitere vorstellbar, die wenigstens die Eigenschaften aus Abschn. 2.1.5 haben. Eine davon ist, eine Markierung einer Stelle als geordnete Liste von Marken aufzufassen, wobei sowohl Hinzufügen als auch Entnehmen an einer Seite geschieht (im Gegensatz zur Sequenz). Wir definieren $\mathfrak{St} \subseteq \mathfrak{A}$ als eine Algebra $\mathfrak{St}(A) = (B, \langle \varepsilon, \oplus, \ominus \rangle)$ mit

$$B = A^* \tag{3.11a}$$

$$\forall a \in A : u(a) = (a) \tag{3.11b}$$

Auch die Basismenge B dieser Algebra ist die Menge aller endlichen Sequenzen über der Menge A. Für die Operationen $\oplus$ und $\ominus$ gilt mit $x, y \in B$ weiterhin

$$(x \oplus y) \ominus x = y \tag{3.12}$$

Das heißt, eine Teilsequenz y kann von einer Sequenz $z = x \oplus y$ nur entfernt werden, wenn y Präfix von z ist. Für $\varepsilon \in B$, $\oplus$ und $\ominus$ definieren wir:

$$\varepsilon = () \tag{3.13a}$$

$$x \oplus y = x \circ y \tag{3.13b}$$

$$x \ominus y = \begin{cases} z & \text{wenn } \exists z \in B : y \circ z = x \\ \text{nicht definiert} & \text{sonst} \end{cases} \tag{3.13c}$$

Auch hier ist mit Gl. (3.13a) die leere Sequenz das neutrale Element. Jede aus Teilsequenzen zusammengesetzte Sequenz ist auch hier Element der Algebra (3.13b). Aber Gl. (3.13c) bestimmt, dass die Operation $\ominus$ nur dann zulässig ist, wenn y Präfix von x ist.

In Petrinetzen benötigt man häufig nicht nur die Markierung einer Stelle, sondern die Markierung des gesamten Netzes. Dies ist dann eine Abbildung (Vektor) $M : P \to \mathfrak{A}(A)$, wobei P die Menge der Stellen eines Netzes und $\mathfrak{A}(A)$ die entsprechende Markierungsalgebra (bzw. ihre Grundmenge) bezeichnet. Die Operationen $\underline{\varepsilon}$, $\underline{\oplus}$, $\underline{\ominus}$ werden für derartige Abbildungen elementweise auf die Operationen ε, $\oplus$, $\ominus$ der Markierungsstruktur zurückgeführt. Es gilt für $\underline{\varepsilon}, M_1, M_2 : P \to \mathfrak{A}(A)$ und für alle $p \in P$:

$$\underline{\varepsilon}(p) = \varepsilon \tag{3.14}$$

$$(M_1 \underline{\oplus} M_2)(p) = M_1(p) \oplus M_2(p) \tag{3.15}$$

$$(M_1 \underline{\ominus} M_2)(p) = M_1(p) \ominus M_2(p) \tag{3.16}$$

Die Atomarität der Marken in der Markierungsstruktur lässt sich ausnutzen, um die Mächtigkeit einer Markierung, das ist die Anzahl der atomaren Elemente, zu definieren. Sie ist induktiv über der Operation $\oplus$ definiert.

Definition 3.2.1 (Mächtigkeit einer Markierung). *Sei $\mathfrak{A}(A)$ eine Markierungsstruktur. Dann ist die* Mächtigkeit einer Markierung *die Funktion $c : \mathfrak{A}(A) \to \mathbb{N}$, für die für alle $x \in \mathfrak{A}(A)$ gilt:*

$$c(x) = \begin{cases} 0 & \text{wenn } x = \varepsilon \\ 1 & \text{wenn es ein } y \in A \text{ so gibt, dass } u(y) = x \\ c(x_1) + c(x_2) & \text{wenn es } x_1, x_2 \in \mathfrak{A}(A) \text{ so gibt, dass } x_1 \oplus x_2 = x \end{cases}$$

*

Auf Grund der Atomarität der Marken ist die Mächtigkeit einer Markierung eindeutig bestimmt.

3.2.3 Die Schaltregel für klassische Petrinetze

Im Vorgriff auf spätere Ausführungen beschreiben wir hier klassische Petrinetz-Typen und deren Schaltregeln. *Klassische Petrinetz-Typen* sind solche Petrinetz-Typen, die Petrinetze mit Hilfe der beiden Basisdimensionen beschreiben. In einem Petrinetz eines solchen Typs gibt es genau zwei Kantentypen – Kanten zu Transitionen (IN) und Kanten von Transitionen (OUT). Es schaltet genau eine Transition. Dabei wird die Verteilung der Marken in der Umgebung der Transition so verändert, dass eine Stelle,

die mit der Transition über eine IN-Kante verbunden ist, Marken verliert und eine Stelle, die mit der Transition über eine OUT-Kante verbunden ist, Marken gewinnt.

Jeder Kante eines Netzes N wird ein Kantentyp $\varphi : f_N \to \{\text{IN}, \text{OUT}\}$ zugeordnet. In der grafischen Darstellung eines Netzes kennzeichnen wir IN-Kanten durch eine ausgefüllte Pfeilspitze auf der entsprechenden Kante in Richtung Transition und eine OUT-Kante durch eine Pfeilspitze in Richtung Stelle. In einem klassischen Petrinetz N gibt es zwischen einer Stelle und einer Transition nicht mehrere Kanten gleichen Typs. Das heißt, es gilt für alle $f \in F$, dass es kein $f' \in F$ mit $f \neq f'$ so gibt, dass $R(f) = R(f')$ und $\varphi(f) = \varphi(f')$.

Der (2-*dimensionale*) *Petrinetz-Typ* ist ein Paar $(A, \mathfrak{A})$, wobei A die Menge von Marken und $\mathfrak{A}$ die Markierungsstruktur für Netze diesen Typs bezeichnet. Die *Instanz* eines Petrinetz-Typs besteht aus einem Netz N, einer Menge von Marken $A' \subseteq A$, $A' \neq \emptyset$ und einer Kanteninschrift $i : F_N \to (A' \to \mathfrak{A}(A'))$. Die Kanteninschrift i ordnet jeder Kante eine Funktion zu, die für eine Belegung $a \in A'$ zu der entsprechenden Markierungsstruktur ausgewertet wird. Diese Markierung wird durch das Schalten einer Transition mit der Belegung a von einer Stelle in ihrer Umgebung entfernt oder zu ihr hinzugefügt. Wir bezeichnen (N, A', i) als *Petrinetz des Typs* $(A, \mathfrak{A})$ und schreiben auch $(N, A', i) : (A, \mathfrak{A})$. Eine *Markierung* von $(N, A', i) : (A, \mathfrak{A})$ ist dann (wie oben) eine Abbildung $M : P_N \to \mathfrak{A}(A')$. Für eine *initiale Markierung* M_0 von (N, A', i) bezeichnen wir $\Sigma = (N, A', i, M_0)$ als *Petrinetz-System des Typs* $(A, \mathfrak{A})$, und wir schreiben dafür $\Sigma : (A, \mathfrak{A})$.

Die Schaltregel für klassische Petrinetz-Typen lässt sich nun einfach definieren. Wir betrachten ein Petrinetz $(N, A', i) : (A, \mathfrak{A})$, eine Transition $t \in T_N$ und eine Belegung $a \in A'$. Zunächst definieren wir zwei Markierungen ${}^-t_a$ und t_a^+ von $(N, A', i) : (A, \mathfrak{A})$, die die Marken repräsentieren, die von der schaltenden Transition t mit der Belegung a konsumiert bzw. produziert werden, für alle $p \in P_N$ sei

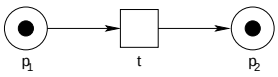
$$ {}^-t_a(p) = \begin{cases} i(f)(a) & \text{wenn } f \in F_N \text{ und } R_N(f) = (p, t) \text{ und } \varphi(f) = \text{IN} \\ \varepsilon & \text{sonst} \end{cases} \tag{3.17} $$

$$ t_a^+(p) = \begin{cases} i(f)(a) & \text{wenn } f \in F_N \text{ und } R_N(f) = (p, t) \text{ und } \varphi(f) = \text{OUT} \\ \varepsilon & \text{sonst} \end{cases} \tag{3.18} $$

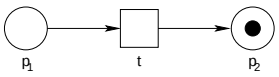
Die klassische Schaltregel ist wie folgt definiert.

Definition 3.2.2 (klassische Schaltregel). *Sei (N, A', i) ein Netz des Typs $(A, \mathfrak{A})$, sei $t \in T_N$ eine Transition, sei $a \in A'$ eine Belegung und seien M_1 und M_2 Markierungen von (N, A', i). Es gilt $M_1 \xrightarrow{t,a} M_2$ genau dann, wenn gilt $M_2 = M_1 \ominus {}^-t_a \oplus t_a^+$.* *

Für den Petrinetz-Würfel [KW98] muss zusätzlich zu einer Definition der Schaltregel wie in Def. 3.2.2 falsches Schalten im Kontakt und mit nicht vorhandenen Marken

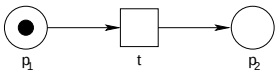


(a) Markierung M_1

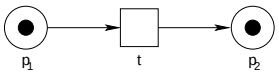


(b) Markierung M_2 nach dem Schalten von t

Abbildung 3.2: Schalten in Kontakt



(a) Markierung M_1



(b) Markierung M_2 nach dem Schalten von t

Abbildung 3.3: Schalten mit einer nicht vorhandenen Marke

ausgeschlossen werden. Hier ist dies nicht nötig, da wir für Marken als Bestandteile von Markierungen Atomarität fordern (siehe Abschn. 2.1.5). Die Operationen $\oplus$ und $\ominus$ der Markierungsstruktur respektieren diese Atomarität.

Die Abbn. 3.2 und 3.3 erläutern die beiden fehlerhaften Schaltvorgänge jeweils für ein Elementares Netzsystem (Petrinetz-Typ $(\{\bullet\}, \mathfrak{Mg})$). Abbildung 3.2 stellt das Schalten der Transition t im Kontakt dar. Die Markierung einer Stelle wird im Petrinetz-Würfel [KW98] als Menge dargestellt. Damit gilt wie üblich auch $\{\bullet\} \cup \{\bullet\} = \{\bullet\}$. Das heißt, beim Schalten von t in Abb. 3.2(a) geht die neu auf p_2 erzeugte Marke in der bereits vorhandenen Marke auf. Mit unserer Definition der Markierungsstruktur ist ein Schalten wie in Abb. 3.2 dargestellt nicht möglich, da $M_1(p_2) \ominus {}^{-}t_a(p_2) \oplus t_a^{+}(p_2) = \{\bullet\} \ominus \{\} \oplus \{\bullet\} = \{\bullet\} \oplus \{\bullet\}$ nach Gl. (3.7b) nicht definiert ist.

Abbildung 3.3 stellt das Schalten mit nicht vorhandenen Marken dar. Im Petrinetz-Würfel [KW98] wird das Entfernen von Marken von einer Stelle mit Hilfe derselben Operation definiert wie der für das Hinzufügen von Marken. Da die Mengenvereinigung im Allgemeinen nicht eindeutig ist ($\{\bullet\} \cup \{\bullet\} = \{\bullet\} \cup \emptyset = \{\bullet\}$), kann mit einer Schaltregel wie in Def. 3.2.2 das Schalten mit nicht vorhandenen Marken nicht ausgeschlossen werden. Die Transition t in Abb. 3.3(a) schaltet so, dass für p_1 gilt $\{\bullet\} \cup \{\bullet\} = \{\bullet\}$. Mit dem Petrinetz-Hyperwürfel ist ein derartiges Schalten ausgeschlossen, da $M_1(p_1) \ominus {}^{-}t_a(p_1) \oplus t_a^{+}(p_1) = \{\bullet\} \ominus \{\bullet\} \oplus \{\} = \{\}$ nach Gl. (3.7c) und nicht wie in $M_2(p_1) = \{\bullet\}$.

3.3 Label

In Abschn. 3.1 wurden Petrinetz-Graphen vorgestellt. Und in Abschn. 3.2 wurden „klassische“ Petrinetze klassifiziert. Beliebige Petrinetze unterscheiden sich von Petri-

netz-Graphen auf den ersten Blick in der schier unerschöpflichen Menge von Beschriftungen. Diese Beschriftungen folgen dabei einem Muster, das vom Typ eines Petrinetzes abhängt. Die Menge der Beschriftungen eines Netzes werden auf Grund ihrer Funktionalität unterteilt. Gleichartige Beschriftungen werden dabei immer gleichartigen Netzelementen zugeordnet. Beschriftungen haben einen Wert, eben das Element, mit dem ein Netzelement oder das Netz beschriftet ist. Wir nennen Beschriftungen *Label* und definieren sie wie folgt.

Definition 3.3.1 (Label). *Sei $N = (P, T, F; R)$ ein Netz und X eine Menge. Dann ist für $Y \in \{P, T, F, \{N\}\}$ eine Abbildung $L : Y \to X$ ein* Label *von Y im Netz N.* $_*$

Label sind zunächst die Elemente, die in Petrinetzen vorkommen und nicht Netzelemente sind. In grafischen Darstellungen von Petrinetzen wird ein Label eines Netzelementes durch eine Zeichenkette oder eine besondere grafische Darstellung gekennzeichnet. Beispiele sind Namen von Netzelementen (wie im Netz aus Abb. 2.1) für Zeichenketten oder Signalkanten (wie im Netz aus Abb. 2.6) für eine grafische Darstellung. Signalkanten eines Netzes N sind in erster Annäherung eine Abbildung $L_{\text{Signal}} : T_N \to \mathcal{P}(T_N)$ von Signal empfangenden Transitionen in eine Menge von Signal sendenden Transitionen. Genauer betrachtet umfasst der Wertebereich eines Labels explizit auch die Mengen, die das Netz bilden. Unter den Begriff des Labels fallen auch Markierungen von Stellen, Kanteninschriften und Kantentypen (siehe Abschn. 3.4).

Tatsächlich hilft uns diese Sichtweise, Phänomene von Labeln in Petrinetzen miteinander zu vergleichen. Zunächst können wir statische Label von dynamischen unterscheiden. Statische Label eines Petrinetzes werden nie verändert, während dynamische Label durch Schalten bzw. andere Überführungsfunktionen jeweils andere Abbildungen mit dem gleichen Definitionsbereich werden. Markierungen von Stellen eines Netzes und Uhren in zeitbehafteten Petrinetzen sind Beispiele für dynamische Label. Markierungen werden beim Schalten verändert, und Uhren laufen unabhängig vom Schalten oder werden beim Schalten zurückgesetzt. Obwohl wir Markierungen als dynamische Label auffassen können, werden wir sie aufgrund ihrer fundamentalen Bedeutung für alle Petrinetze in den folgenden Abschnitten gesondert behandeln. Markierungen werden erst dann wieder wie andere Label behandelt, wenn wir vom Schalten eines Petrinetzes abstrahieren wie in Kap. 4.

Viele Label in Petrinetzen sind statische Label. Ihre Werte bleiben konstant. Statische Label unterteilen wir zum einen nach ihrer Funktion bzgl. des Schaltens und zum anderen nach der Mächtigkeit ihres Wertebereiches. Label sind entweder schaltrelevant oder nicht. Das heißt, ihre Werte beeinflussen die Aktivierung einer Transition (oder eines Schrittes), oder ihre Werte werden auf Grund anderer Eigenschaften von Petrinetzen oder ihres praktischen Einsatzes vergeben. Label letzterer Art sind beispielsweise Kennzeichnungen von Modellstrukturen, Namen von Netzelementen oder

die initiale Markierung von Stellen. Solche Label werden uns im Folgenden nicht weiter interessieren.

Ein schaltrelevantes Label dagegen definiert ein Prädikat, das eine weitere Schaltbedingung für eine Transition (bzw. eine Belegung, siehe Abschn. 3.5.1) oder einen Schritt (siehe Abschn. 3.5.2) aufgrund des eigenen Wertes und der Werte bestimmter anderer Label ist. Beispiele derartiger schaltrelevanter statischer Label sind *transition guards* (zusätzliche Aktivierungsbedingungen von Transitionen), Kanteninschriften, Kapazitätsanschriften, Zeitanschriften, Kantentypen usw. Manche Label wie Kanteninschriften und Kantentypen haben eine so fundamentale bzw. komplexe Bedeutung für das Schalten in Petrinetzen, dass sie in den folgenden Abschnitten gesondert behandelt werden.

Meist werden Label durch Zeichenketten ausgedrückt, die Netzelementen zugeordnet werden. Manchmal, insbesondere bei sehr kleinen Wertebereichen, ist es üblich, die Werte der Label grafisch (z. B. durch Farbe und Gestaltung) anzuzeigen.

In einem konkreten Petrinetz werden lediglich die Werte der Label festgelegt. Der Wertebereich der Label, die schaltrelevanten Prädikate und die Überführungsfunktionen dynamischer Label werden vom Petrinetz-Typ unabhängig von konkreten Netzen festgelegt. Diese Festlegung gilt dann für jede Instanz des Petrinetz-Typs, das heißt, für jedes konkrete Petrinetz eines bestimmten Petrinetz-Typs. Die Festlegung eines Labels durch einen Petrinetz-Typ nennen wir Universum des Labels.

Schaltrelevante Label haben ein Aktivierungsprädikat v. Schaltrelevante Label von Stellen eines Netzes haben außerdem (bzw. meist stattdessen) ein Effektprädikat k. Das Aktivierungsprädikat bestimmt, ob ein Label in einem Schaltmodus (siehe Def. 3.5.2), abhängig von seinem Wert bzw. den Werten weiterer Label, gültig ist oder nicht. Das Effektprädikat eines Labels einer Stelle bestimmt, ob eine Markierung bzgl. des Labelwertes gültig ist oder nicht.

Dynamische Label zeichnen sich vor allem durch Überführungsfunktionen aus. Auch die Definition einer Überführungsfunktion ist nicht von einem konkreten Petrinetz abhängig sondern vom Petrinetz-Typ. Eine Überführungsfunktion eines Labels verändert die Abbildung, die das Label definiert. Definitions- und Wertebereich der Abbildung bleiben gleich. Es gibt zwei Arten von Überführungsfunktionen. Eine wird beim Schalten eines Schrittes (siehe Abschn. 3.5.4) angewandt, die andere, unabhängig von einem Schritt (siehe Abschn. 3.5.5), z. B. statt des Schaltens eines Schrittes. Ein Beispiel für eine Überführungsfunktion der ersten Art ist das Zurücksetzen von Uhren in Zeit-Petrinetzen. Ein Beispiel für eine Überführungsfunktion der zweiten Art ist ein Uhrentick in Zeit-Petrinetzen.

Das Aktivierungsprädikat v ist für einen Wertebereich X und eine Menge von Schaltmodi Θ (siehe Abschn. 3.5.1) wie folgt definiert

$$v : X \to (\Theta \to \mathbb{B}) \tag{3.19}$$

Das Effektprädikat k für Label von Stellen ist für einen Wertebereich X, eine Markenmenge A und eine Markierungsstruktur $\mathfrak{A}$ definiert

$$k : X \to (\mathfrak{A}(A) \to \mathbb{B}) \tag{3.20}$$

Wir definieren nun den Begriff des Universums eines Labels.

Definition 3.3.2 (Universum eines Labels). *Seien N ein Netz, X eine beliebige Menge, Θ eine Menge von Schaltmodi, A eine Markenmenge und $\mathfrak{A}$ eine Markierungsstruktur, dann ist das Tupel $\mathfrak{l} = (X, v, k, \overset{\widehat{s}}{\rightarrowtail}, \rightarrowtail)$ das Universum eines Labels, wenn gilt*

- *es gibt ein $Y \in \{P_N, T_N, F_N, \{N\}\}$ so, dass $L : Y \to X$ ein Label von Y in N ist*
- *v eine Abbildung $v : X \to (\Theta \to \mathbb{B})$ ist*
- *k eine Abbildung $k : X \to (\mathfrak{A}(A) \to \mathbb{B})$ ist*
- *$\overset{\widehat{s}}{\rightarrowtail}$ eine Überführungsfunktion eines Labels $L : Y \to X$ in ein Label $L' : Y \to X$ (geschrieben $L \overset{\widehat{s}}{\rightarrowtail} L'$) ist*
- *$\rightarrowtail$ eine Überführungsfunktion eines Labels $L : Y \to X$ in ein Label $L'' : Y \to X$ (geschrieben $L \rightarrowtail L''$) ist.*

Die Label L, L', L'' heißen Label des Universums $\mathfrak{l}$. *

Für triviale Fälle sind einige Elemente des entsprechenden Universums eines Labels trivial. Einen Wertebereich hat jedes Label in seinem Universum. Ein triviales Aktivierungsprädikat bzw. Effektprädikat[1] liefert für jeden Wert den Wahrheitswert wahr. Eine triviale Überführungsfunktion liefert die identische Abbildung. So ist das Aktivierungsprädikat v für triviale Fälle für alle $x \in X$ und für alle $\theta \in \Theta$ so, dass $v(x)(\theta) =$ wahr. Das triviale Effektprädikat k ist für alle $x \in X$ und für alle $m \in \mathfrak{A}(A)$ so, dass $k(x)(m) =$ wahr. Schließlich sind die trivialen Überführungsfunktionen $\overset{\widehat{s}}{\rightarrowtail}$ und $\rightarrowtail$ für ein Label L so definiert, dass $L \overset{\widehat{s}}{\rightarrowtail} L$ und $L \rightarrowtail L$ gilt. Wir benutzen für diese trivialen Fälle die folgenden Kurzschreibweisen: v_{def} für ein triviales Aktivierungsprädikat, k_{def} für ein triviales Effektprädikat und für die trivialen Überführungsfunktionen $\overset{\widehat{s}}{\rightarrowtail}_{\text{def}}$ und $\rightarrowtail_{\text{def}}$. Ein Label, das sowohl ein triviales Aktivierungsprädikat als auch ein triviales Effektprädikat als auch triviale Überführungsfunktionen hat, ist ein nicht schaltrelevantes Label.

[1] Für die parametrisierte Schaltregel (Abschn. 3.5) sind nur Effektprädikate von Stellen-Labeln relevant.

Das Universum eines Labels ist Bestandteil des Petrinetz-Typs. Die Menge der Labeluniversen eines Petrinetz-Typs bezeichnen wir mit $\mathfrak{L}$. Zu einem Netz N dieses Typs gehört eine Menge von (initialen[2]) Labeln $\mathcal{L}$, so dass für alle $L \in \mathcal{L}$ ein $\mathfrak{l} \in \mathfrak{L}$ so existiert, dass L ein Label des Universums $\mathfrak{l}$ ist. Das heißt, die Menge der Label $\mathcal{L}$ setzt sich wie folgt zusammen

$$\mathcal{L} = \left\{ L \mid L : Y \to \mathfrak{l}_{(1)} \text{ mit } Y \in \{P_N, T_N, F_N, \{N\}\} \text{ und } \mathfrak{l} \in \mathfrak{L} \right\} \tag{3.21}$$

Jedes Petrinetz N hat wenigstens drei Label. Ein Label M beschreibt die Markierung jeder Stelle für ein Netz N mit der Markenmenge A und der Markierungsstruktur $\mathfrak{A}$.

$$M : P_N \to \mathfrak{A}(A) \tag{3.22}$$

Jeder Stelle eines Netzes ist also eine Markierung zugewiesen, die auch das neutrale Element ε der Markierungsalgebra $\mathfrak{A}(A)$ sein kann.

Kanten haben ein Label i für die Kanteninschrift. Im Allgemeinen ist i ein Label von Kanten im Netz N mit der Markenmenge A und der Markierungsstruktur $\mathfrak{A}$. Die Menge $\mathcal{X}$ bezeichnet freie Variablen, die vom Schaltmodus (siehe Abschn. 3.5.1) einer Transition belegt werden.

$$i : F_N \to \mathfrak{A}(A \cup \mathcal{X}) \quad \text{mit } A \cap \mathcal{X} = \emptyset \tag{3.23}$$

Auch hier kann es Kanteninschriften geben, die in das leere Element ε der Markierungsalgebra auswerten. Schaltmodi, die dies zur Folge haben, werden tatsächlich in manchen *high level* Netzen benötigt.

Und schließlich haben in unserem Ansatz alle Kanten eines Petrinetzes ein Label φ, das ihren Kantentyp bestimmt (vgl. Abschn. 3.4). Für ein Netz N und eine Menge von Kantentypen Φ ist

$$\varphi : F_N \to \Phi \tag{3.24}$$

ein Label von Kanten im Netz N.

Da diese drei Label (Markierung, Kanteninschrift, Kantentyp) eine besondere Rolle spielen, definieren wir für sie keine Labeluniversen.

3.4 Kantentypen

In Abschn. 3.2.3 wurden Kantentypen deklariert, um den Kanten eines Petrinetzes eine Richtung zu geben. Damit konnten einer Transition eindeutig die Stellen ihrer

[2] Statische Label eine Petrinetzes verändern sich nicht, während für dynamische Label (nur) ihr initialer Zustand angegeben wird.

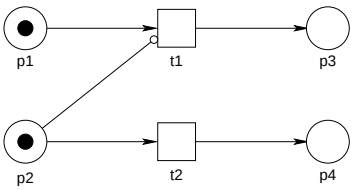
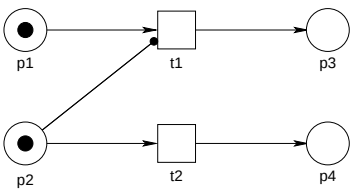
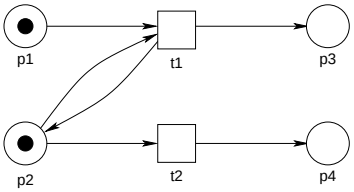


Abbildung 3.4: S/T-Netz mit Inhibitorkante



(a) S/T-Netz mit Testkante



(b) S/T-Netz mit Schlinge

Abbildung 3.5: Testkante vs. Schlinge

Umgebung zugeordnet werden, von denen sie beim Schalten Marken konsumiert bzw. auf denen sie Marken produziert. In diesem Abschnitt werden wir zunächst einige weitere Kantentypen betrachten und später klassifizieren. Die verschiedenen Kantentypen werden benötigt, um spezielles Schaltverhalten von Transitionen zu modellieren.

Abbildung 3.4 zeigt ein S/T-Netz mit einer Inhibitorkante [Pet77a] zwischen p2 und t1. Grafisch wird eine Inhibitorkante durch einen offenen Kreis auf der Kante an ihrer Transition dargestellt. Die Transition t1 kann schalten, wenn alle Bedingungen für das Schalten von Transitionen in S/T-Netzen (ohne Inhibitorkanten) erfüllt sind *und* wenn p2 nicht markiert ist. Eine Inhibitorkante wird verwendet, um Prioritätsrelationen bzgl. des Schaltens von Transitionen zu modellieren. Jonathan Billington erweitert den Begriff der Inhibitorkante um die Abwesenheit der Marken, die durch die Kanteninschrift ausgedrückt werden [Bil91].

Abbildung 3.5(a) zeigt ein S/T-Netz mit einer Testkante (auch *read arc* oder *condition arc* genannt) [VSY98, HL00] zwischen t1 und p2. Eine Testkante wird grafisch durch einen gefüllten Kreis auf der Kante an ihrer Transition dargestellt. Die Transition t1 kann nur schalten, wenn auf p2 wenigstens eine Marke vorhanden ist. Die Testkante drückt aus, dass eine Marke zum Schalten vorhanden sein muss, aber von der Transition nicht konsumiert wird, also permanent auf der Stelle bleibt. Für S/T-Netze kann man Testkanten gewöhnlich durch eine Schlinge wie in Abb. 3.5(b) ersetzen. In einem Netz, dessen Schaltregel es zulässt, dass mehrere Transitionen in einem Schritt (gleichzeitig) schalten, kann eine Testkante nicht durch eine Schlinge ersetzt werden, da sich dann ein anderes Schaltverhalten ergibt. Die Transitionen t1 und t2 in Abb. 3.5(b) stehen um die Marke auf p2 im Konflikt zueinander. In einem Schritt

kann nur eine von beiden konzessionierten Transitionen schalten. Im Gegensatz dazu können für ein Netz mit Testkanten wie in Abb. 3.5(a) mit der Schrittschaltregel die Transitionen t1 und t2 gemeinsam in einem Schritt schalten, da die Transitionen t1 und t2 nicht im Konflikt um die Marke auf p2 stehen.

Es werden weitere Kanten mit jeweils verschiedener Bedeutung beschrieben [LC94]. Einige davon werden im Folgenden kurz erläutert. Eine Abräumkante (*clear arc*) leert beim Schalten ihrer Transition ihre Stelle, eine Verdopplungskante (*double arc*) erzeugt genau so viele Marken, wie beim Schalten ihrer Transition von der Stelle gelesen und unverändert gelassen werden. Schließlich wird eine Setzkante (*set arc*) beschrieben, die beim Schalten ihrer Transition ihre notwendig leere Stelle mit einer Markierung belegt.

3.4.1 Klassifikation von Kantentypen

In diesem Abschnitt wird ein neuer Ansatz zur Klassifikation von Kantentypen in Petrinetzen erläutert. Der Ansatz beruht auf der Beobachtung, dass eine Kante eine Doppelfunktion hat. Sie bestimmt die Aktivierung ihrer Transition (mit Hilfe der Markierung ihrer Stelle) mit und sie bestimmt den Effekt der Transition auf die Markierung ihrer Stelle beim Schalten ihrer Transition. Wir trennen diese beiden Funktionen voneinander und betrachten sie separat. Einem Kantentyp ist dann eine Aktivierungsbedingung und ein Effekt zugeordnet.

Im Folgenden wird der Typ einer Kante in Abbildungen durch besondere grafische Elemente dargestellt. Die Darstellung eines Kantentyps setzt sich zusammen aus einer grafischen Darstellung der Aktivierungsbedingung und einer grafischen Darstellung des Effektes. Die Aktivierungsbedingung der Kante wird durch ein grafisches Element (Pfeilspitze, Strich, Kreis) an ihrer Transition wiedergegeben. Der Effekt einer Kante wird durch ein grafisches Element an ihrer Stelle dargestellt.

Tabelle 3.1 kombiniert je eine Aktivierungsbedingung mit je einem Effekt zu einem Kantentyp. Die Spalten der Tabelle stellen die Aktivierungsbedingungen dar, und die Zeilen stellen Effekte der Kanten dar. Ein Kantentyp ergibt sich dann als der entsprechende Eintrag in der Matrix der Tabelle. Für jede Aktivierungsbedingung ist eine charakterisierende Bezeichnung (erste Zeile des Tabellenkopfes), eine formale Darstellung der Bedingung (zweite Zeile) und eine grafische Repräsentation (dritte Zeile) angegeben. Ebenso ist für jeden Effekt eine charakterisierende Bezeichnung (erste Spalte der Tabelle), eine formale Darstellung des Effektes (zweite Spalte) und eine grafische Repräsentation (dritte Spalte) angegeben.

Die formale Darstellung der Aktivierungsbedingung bezeichnet die Markierung der Stelle der Kante mit m und die Markierung mit i, die sich aus der Auswertung der Kanteninschrift ergibt. Außerdem werden die Operationen $\oplus$ und $\ominus$ sowie die Konstante ε (für die leere Markierung) der Markierungsstruktur (vgl. Abschn. 3.2.2) verwendet. In der formalen Darstellung des Effektes wird die Änderung der Markierung der Stelle

Tabelle 3.1: Klassifikation der Kantentypen in Petrinetzen

Char.	Effekt	Char. Aktivierungsbedingung Grafik	POT.OUT $m \oplus i$ ist def.	POT.IN $m \ominus i$ ist def.	NO keine	EQUAL $m = i$	ZERO $m = \varepsilon$	PUT-INHIB $m \oplus i$ ist n. def.	INHIB $m \ominus i$ ist n. def.
TEST	kein		pot. OUT	READ	unabh.	EQUAL	ZERO TEST	PUT INHIBITOR	INHIBITOR
ADD	$\oplus i$		OUT	DOUBLE?	ADD?	strict DOUBLE?	compos. SET	—	INHIB ADD?
SUB	$\ominus i$		POT.OUT SUB?	IN	SUB?	preserv. CLEAR	ZERO SUB?	PUT-INHIB SUB?	—
SET	i		POT.OUT SET	POT.IN SET	uncond. SET	preserv. SET	SET	PUT-INHIB SET	INHIB SET
CLEAR	ε		POT.OUT CLEAR	TEST CLEAR	uncond. CLEAR	CLEAR	ZERO CLEAR	PUT-INHIB CLEAR	INHIB CLEAR
REFRESH	m		POT.OUT REFRESH	POT.IN REFRESH	REFRESH	EQUAL REFRESH	ZERO REFRESH	PUT-INHIB REFRESH	INHIB REFRESH

Legende:

? die Durchführung des Effektes wird von der Aktivierungsbedingung nicht gesichert

— Kante existiert nicht wg. Widerspruch zwischen Aktivierungsbedingung und Effekt

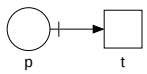


Abbildung 3.6: READ-Kante eines Netzes

beim Schalten der Transition der Kanten angegeben ($\oplus i$ bzw. $\ominus i$), oder es wird der Wert angegeben auf den die Markierung verändert wird (i, ε oder m). Für einen konkreten Petrinetz-Typ gilt $m, i \in \mathfrak{A}(A)$, wobei $\mathfrak{A}$ die Markierungsstruktur und A die Markenmenge des Petrinetz-Typs bezeichnet.

Die grafische Darstellung der Aktivierungsbedingung und des Effektes einer Kante orientiert sich an bekannten Darstellungen von Kantentypen. Das lässt sich aber mit der strikten Trennung von Aktivierungsbedingung und Effekt sowie ihrer jeweiligen Darstellung nicht konsequent durchhalten. Wir stellen wenigstens die gebräuchlichsten Kantentypen IN und OUT wie gewohnt als Pfeile zwischen Stelle und Transition mit schwarzer Pfeilspitze dar. Wir nennen diese Kantentypen gebräuchlich, da jeder Petrinetz-Typ zwei Arten von Kanten zulässt. Kanten der einen Art verlaufen von Stellen zu Transitionen (IN) und Kanten der anderen Art von Transitionen zu Stellen (OUT). IN-Kanten zeigen an, dass beim Schalten der Transition Marken von der Stelle konsumiert werden. OUT-Kanten zeigen an, dass Marken auf der Stelle produziert werden.

Daraus ergeben sich die grafischen Darstellungen für die Aktivierungsbedingungen POT.OUT und POT.IN sowie für die Effekte ADD und SUB. Für den Effekt TEST wählen wir eine intuitive Darstellung, die anzeigt, dass sich die Markierung der Stelle der Kante *nicht* ändert. Und für die Aktivierungsbedingung NO zeigen wir grafisch, dass die Aktivierung der Transition *nicht* von der entsprechenden Kante abhängt. Alle anderen grafischen Darstellungen der Aktivierungsbedingung und des Effektes sind willkürlich. Lediglich die Darstellung der Aktivierungsbedingung INHIB orientiert sich an gebräuchlichen Darstellungen für eine Inhibitorkante [Pet77a].

Abbildung 3.6 zeigt als Beispiel eine READ-Kante zwischen der Stelle p und der Transition t. Eine READ-Kante entspricht einer Testkante (vgl. Abb. 3.5(a)). Die Pfeilspitze der READ-Kante, die Aktivierungsbedingung der Kante darstellend, symbolisiert, dass wenigstens eine Marke auf p vorhanden sein muss, wenn t schaltet. Der Strich auf der Kante in der Nähe von p symbolisiert, dass auf diese Stelle kein Effekt ausgeführt wird, sich die Markierung der Stelle durch das Schalten von t nicht ändert.

In der Matrix der Tab. 3.1 sind einige Zellen mit einer grafischen Darstellung des Kantentyps belegt, wie sie sich aus der Kombination der grafischen Darstellung der Aktivierungsbedingung (Spalte) und des Effektes (Zeile) ergibt. Es sind die Kantentypen als Grafik dargestellt, bei denen sich Aktivierungsbedingung und Effekt auf keinen Fall widersprechen. Für alle mit „?“ gekennzeichneten Kantentypen gilt, dass die Aktivierungsbedingung der entsprechenden Kante nicht von ihrem Effekt „gesi-

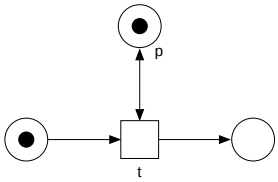


Abbildung 3.7: Elementares Netzsystem mit DOUBLE-Kante

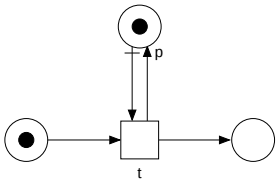


Abbildung 3.8: Elementares Netzsystem aus Abb. 3.7 in Ersatzdarstellung

chert“ wird. Das heißt, es kann Bedingungen geben, in denen die Transition einer solchen Kante aktiviert ist, aber der Effekt auf die Markierung ihrer Stelle dennoch nicht zulässig ist.

Wir betrachten dazu das Beispiel in Abb. 3.7. Es stellt ein Elementaren Netzsystemen vergleichbares Netz mit einer DOUBLE-Kante dar. Das heißt, für die Markierungsmenge (vgl. Abschn. 3.2.1) gilt $A = \{\bullet\}$, und die Markierungsstruktur ist eine Menge ($\mathfrak{A} = \mathfrak{Mg}$). Damit gilt für $\mathfrak{A}(A) = (\{\varepsilon, \bullet\}, \langle \varepsilon, \oplus, \ominus \rangle)$ mit den entsprechenden Regeln für $\mathfrak{Mg}$ (siehe Abschn. 3.2.2). Entsprechend der Klassifikation aus Tab. 3.1 ist die Kante zwischen der Stelle p und der Transition t vom Typ DOUBLE. Deshalb hat die grafische Darstellung der Kante jeweils eine Pfeilspitze an p und t. Die Pfeilspitze an t stellt die Aktivierungsbedingung POT.IN, und die Pfeilspitze an p stellt den Effekt ADD dar. Die Aktivierungsbedingung des Kantentyps überprüft, ob die Operation $\ominus$ auf der Markierung der Stelle p ($\{\bullet\}$) und der resultierenden Markierung der Kanteninschrift (in diesem Fall $\{\bullet\}$) definiert ist. Dies ist hier der Fall. Die Aktivierungsbedingungen der anderen beiden Kanten, zu denen die Transition t gehört, sind ebenfalls erfüllt. Dennoch ist t nicht aktiviert, da der Effekt auf die Stelle p beim Schalten von t nicht zulässig ist, denn die Operation $\oplus$ ist für $\{\bullet\} \oplus \{\bullet\}$ hier nicht definiert.

Wir ersetzen daher eine solche Kante eines fraglichen Typs durch zwei Kanten. Die Aktivierungsbedingung der zu ersetzenden Kante wird durch eine Kante ohne Effekt (Zeile TEST in Tab. 3.1) umgesetzt, die die gleiche Aktivierungsbedingung hat. Der Effekt der zu ersetzenden Kante wird durch eine Kante umgesetzt, deren Aktivierungsbedingung ihrem Effekt entspricht. Für das Beispiel aus Abb. 3.7 ergibt sich damit die Ersatzdarstellung in Abb. 3.8. Nun ist die Transition t schon auf Grund der Akti-

vierungsbedingung der OUT-Kante zwischen p und t nicht aktiviert, da die Bedingung „$\{\bullet\} \oplus \{\bullet\}$ ist definiert“ nicht erfüllt ist.

Der Ersatzdarstellung liegt die folgende allgemeine Beobachtung zu Grunde. Der Effekt auf eine Stelle, die mehrere adjazente TEST-Kanten und genau eine adjazente Kante hat, die nicht TEST-Kante ist, hängt nur von dieser Nicht-TEST-Kante ab. Wenn die Aktivierungsbedingung dieser Nicht-TEST-Kante Voraussetzung für die mit ihrem Effekt verbundene Operation der Markierungsstruktur ist und wenn die Aktivierungsbedingungen aller Kanten erfüllt sind, dann ist der Effekt auf der Stelle zulässig.

Ohne Beschränkung der Allgemeinheit können wir auch alle weiteren in Tab. 3.1 nicht als Grafik dargestellten Kantentypen ersetzen. Eine Kante eines solchen Typs ersetzen wir durch zwei Kanten. Die eine Ersatzkante ist eine TEST-Kante mit derselben Aktivierungsbedingung wie die zu ersetzende Kante. Die andere Ersatzkante ist eine Kante, deren Typ denselben Effekt hat, wie die zu ersetzende Kante und die in der Matrix der Tab. 3.1 eine grafische Darstellung besitzt. Die Kanteninschrift der zu ersetzenden Kante wird an beide Ersatzkanten übertragen.

Wir klassifizieren die Kantentypen aus Tab. 3.1 nun entsprechend ihrer Schaltsemantik. Wenn eine Transition schaltet, die über eine TEST-Kante mit einer Stelle verbunden ist, dann wird die Markierung der Stelle durch diese Kante nicht verändert, aber gelesen. Wir werden solche Kanten der Klasse *read* zuordnen. Eine SUB-Kante bewirkt beim Schalten ihrer Transition, dass Marken der Stelle gelöscht werden. Wir bezeichnen die Klasse solcher Kanten mit *delete*. Eine ADD-Kante bewirkt beim Schalten ihrer Transition, dass Marken zu der Markierung der Stelle hinzugefügt werden. Die Klasse dieser Kanten bezeichnen wir mit *insert*. Und schließlich fassen wir die Kanten mit den Effekten SET, CLEAR und REFRESH, die einen exklusiven verändernden Zugriff auf die Markierung der Stelle beim Schalten ihrer Transition benötigen, in der Klasse *write* zusammen.

Für die tatsächlich verwendeten Kantentypen (also die 11 Kantentypen mit einer grafischen Darstellung) aus Tab. 3.1 führen wir mathematische Notationen ein. Die Zuordnung der verbalen Beschreibung zu ihrer mathematischen Notation zeigt Tabelle 3.2.

Tabelle 3.3 zeigt die Zuordnung der Kantentypen zu entsprechenden Teilmengen der Menge aller Kantentypen. Wir bezeichnen die Menge aller Kantentypen mit Φ_{all}. Es sei $\Phi_{\text{all}} = \Phi_{\text{r}} \cup \Phi_{\text{d}} \cup \Phi_{\text{i}} \cup \Phi_{\text{w}}$. Eine Kante mit einem Typ aus der Menge der *read*-Kanten Φ_{r} lässt beim Schalten ihrer Transition die Markierung ihrer Stelle unverändert. Die Markierung der Stelle wird jedoch benötigt, um die Aktivierung der Transition entsprechend der Semantik der Kante zu bestimmen. Eine Kante vom Typ *delete* bestimmt ihrer Transition, Marken von der Stelle der Kante zu konsumieren. Eine Kante vom Typ *insert* lässt ihre Transition Marken auf ihrer Stelle produzieren. Und eine Kante, deren Typ mit *write* klassifiziert wird, lässt ihre Transition die Markierung ihrer Stelle komplett neu setzen.

Tabelle 3.2: Zuordnung der verbalen Kantentypbezeichner aus Tabelle 3.1 zu mathematischen Notationen

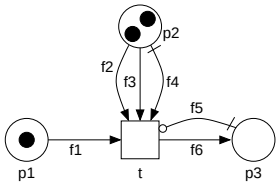
verbale Bezeichnung	math. Notation
pot. OUT	$\mathcal{F}_{\mathrm{pO}}$
READ	$\mathcal{F}_{\mathrm{RD}}$
EQUAL	$\mathcal{F}_{\mathrm{E}}$
ZERO TEST	$\mathcal{F}_{\mathrm{ZT}}$
PUT INHIBITOR	$\mathcal{F}_{\mathrm{PI}}$
INHIBITOR	$\mathcal{F}_{\mathrm{I}}$
IN	$\mathcal{F}_{\mathrm{IN}}$
OUT	$\mathcal{F}_{\mathrm{OUT}}$
uncond. SET	$\mathcal{F}_{\mathrm{S}}$
uncond. CLEAR	$\mathcal{F}_{\mathrm{C}}$
REFRESH	$\mathcal{F}_{\mathrm{RF}}$

Tabelle 3.3: Klassifikation der Kantentypen nach ihrer Schaltsemantik

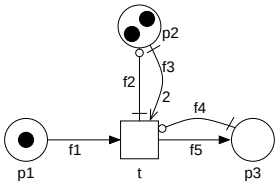
Klassifikation	math. Notation		Kantentypen (math. Notation)
read	Φ_{r}	$=$	$\{\mathcal{F}_{\mathrm{pO}}, \mathcal{F}_{\mathrm{RD}}, \mathcal{F}_{\mathrm{E}}, \mathcal{F}_{\mathrm{ZT}}, \mathcal{F}_{\mathrm{PI}}, \mathcal{F}_{\mathrm{I}}\}$
delete	Φ_{d}	$=$	$\{\mathcal{F}_{\mathrm{IN}}\}$
insert	Φ_{i}	$=$	$\{\mathcal{F}_{\mathrm{OUT}}\}$
write	Φ_{w}	$=$	$\{\mathcal{F}_{\mathrm{S}}, \mathcal{F}_{\mathrm{C}}, \mathcal{F}_{\mathrm{RF}}\}$

Kanten mit dem gleichen Klassifikationstyp können bis auf *write*-Kanten leicht kombiniert werden. Ein Problem sind mehrere Kanten verschiedener Klassifikationstypen zwischen einer Stelle und einer Transition, wenn mehrere Kanten nicht vom Typ *read* sind. Eine Transition ist konzessioniert, wenn wenigstens alle Aktivierungsbedingungen ihrer Kanten erfüllt sind. Dies allein genügt jedoch nicht zur Überprüfung der Konzession, wenn mehrere Kanten einen Effekt auf die gleiche Stelle haben. Dafür werden wir den Begriff *kumulierter Effekt* (siehe Abschn. 3.5.2) einführen. In einer von den Klassifikationstypen bestimmten Reihenfolge wird der kumulierte Effekt auf eine Stelle berechnet. Ist dieser Effekt zulässig, kann die Transition schalten.

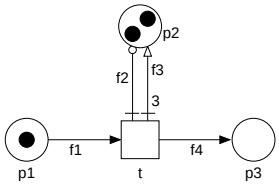
Abbildung 3.9 zeigt Beispiele für zulässige und unzulässige Kombinationen von Kantentypen. Zur besseren Lesbarkeit sind alle Kanten bezeichnet (mit f1, ..., f6). Abbildung 3.9(a) zeigt ein Netz mit Kanten der Typen IN (Kanten f1, f2, f3), READ (f4), INHIBITOR (f5) und OUT (f6). Die Transition t kann schalten, da die Aktivierungs-



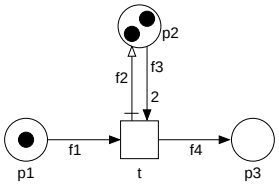
(a) zulässige Kombinationen (*read*/*delete*/*delete* und *read*/*insert*)



(b) zulässige Kombinationen (*read*/*write* und *read*/*insert*)



(c) unzulässige Kombination (*write*/*write*)



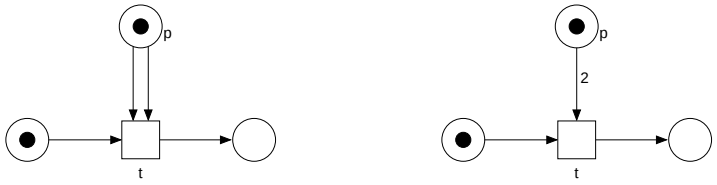
(d) unzulässige Kombination (*write*/*delete*)

Abbildung 3.9: S/T-Netze mit Kanten verschiedener Typen

bedingung aller Kanten erfüllt ist und der kumulierte Effekt auf allen Stellen in der Umgebung von t zulässig ist. Der kumulierte Effekt der Kanten f2, f3 und f4 setzt sich aus dem Effekt der einzelnen Kanten (durch beliebige Hintereinanderausführung der Effekte in beliebiger Reihenfolge) zusammen. Kante f4 hat keinen Effekt auf p2, f2 entfernt eine Marke von p2 und f3 die andere. Kante f6 fügt eine Marke zu p3 hinzu, und f5 hat keinen Effekt auf p3.

Abbildung 3.9(b) zeigt ebenfalls zwei zulässige Kombinationen von Kantentypen. Transition t kann schalten, da die Aktivierungsbedingung aller Kanten im Umfeld von t erfüllt sind. Beim Schalten von t hat nur die Kante f2 (Typ uncond. CLEAR) Effekt auf die Stelle p2 und nur die Kante f5 (OUT) auf die Stelle p3. Die Kanten f3 und f4 sind vom Klassifikationstyp *read*. Die Kante f3 hat eine Kanteninschrift, die anzeigt, dass die EQUAL-Kante genau 2 Marken auf p2 erwartet.

Die Abbn. 3.9(c) und 3.9(d) stellen unzulässige Kombinationen von Kanten dar, da die Kantentypen jeweils auf der Stelle p2 einen unzulässigen kumulierten Effekt erzeugen. Die Aktivierungsbedingungen der Kanten im Umfeld der Transition t in Abb. 3.9(c) sind erfüllt. Der kumulierte Effekt auf p2 lässt einen Nichtdeterminismus zu. Je nachdem in welcher Reihenfolge die Effekte der Kanten f2 (uncond. CLEAR) und f3 (uncond. SET) ausgeführt werden, besteht die resultierende Markierung auf p2 aus 3 schwarzen Marken (Effekt von f3 nach f2) oder keiner Marke (f2 nach f3). Dieser Nichtdeterminismus ist unerwünscht; die Kombination von *write*-Kanten zwischen einer Transition und einer Stelle ist nicht zulässig.



(a) Aktivierungsbedingungen erfüllt, Effekt nicht zulässig

(b) Aktivierungsbedingung nicht erfüllt

Abbildung 3.10: Aktivierungsbedingung und Effekt von Kanten (S/T-Netz)

Auch die Kombination von *write*- und *delete*-Kanten ist nicht zulässig, wie das Beispiel in Abb. 3.9(d) zeigt. Transition t ist aufgrund der Aktivierungsbedingungen der Kanten in ihrem Umfeld aktiviert. Ein Schalten von t würde zu einer nicht definierten Markierung auf p2 führen, wenn der Effekt von f2 (uncond. SET) vor dem Effekt von f3 (IN) ausgeführt wird. Die Kante f2 hat einen Effekt von einer Marke auf p2, der anschließende Effekt von f3 führt aus der Algebra heraus und verletzt die Eigenschaft aus Gl. (3.4c). Wenn jedoch der Effekt von f3 vor dem von f2 ausgeführt wird, ergibt sich eine resultierende Markierung von einer schwarzen Marke auf p2.

Im Allgemeinen genügt es nicht, die Aktivierungsbedingungen der Kanten im Umfeld einer Transition zu überprüfen, um die Konzession der Transition festzustellen, wie Abb. 3.10 an S/T-Netzen zeigt. Die Aktivierungsbedingungen der Kanten im Umfeld der Transition t aus Abb. 3.10(a) sind alle erfüllt. Dennoch ist der kumulierte Effekt der beiden *delete*-Kanten zwischen t und p auf die Stelle p nicht zulässig. Dagegen ist die Aktivierungsbedingung der Kante zwischen t und p in Abb. 3.10(b) bereits nicht erfüllt, so dass t dort nicht konzessioniert ist. Wir wollen mehrere Kanten gleichen Typs nicht ausschließen, da es z. B. in manchen Petrinetz-Werkzeugen durchaus vorkommt, Kantengewichte (also Kanteninschriften) durch mehrfache Kanten auszudrücken.

Jedem Petrinetz N ist eine Menge von Kantentypen Φ zugeordnet.

$$\Phi = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\} \qquad \mathcal{F}_1, \ldots, \mathcal{F}_n \text{ sind paarweise disjunkt} \tag{3.25a}$$

Und es gibt eine eindeutige Abbildung

$$\varphi : F_N \to \Phi \tag{3.25b}$$

die jeder Kante einen Typ zuweist. Für die meisten Petrinetze darf es keine zwei Kanten vom gleichen Typ zwischen derselben Stelle und derselben Transition geben:

$$\text{für alle } f \in F \text{ gibt es kein } f' \in F \text{ mit } f \neq f' \text{ und } R(f) = R(f') \text{ und } \varphi(f) = \varphi(f') \tag{3.26}$$

Eine solche Einschränkung verhindert Netze wie in Abb. 3.10(a); dort haben die Kanten zwischen p und t den gleichen Typ.

3.4.2 Aktivierungsbedingung

Für jeden Kantentyp ist (wie auch aus Tab. 3.1 hervorgeht) eine Aktivierungsbedingung definiert. Sie bezieht sich auf die Markierung der Stelle und die Kanteninschrift der Kante. Der Petrinetz-Typ gewährleistet, dass die Kanteninschrift in ein Element derselben Algebra überführt wird, wie die Algebra, deren Element die Markierung der Stelle ist. Wir definieren für jeden Kantentyp $\mathcal{F}$ aus der Menge Φ_{all} ein Aktivierungsprädikat $\alpha_{\mathcal{F}}$ über Elementen der Grundmenge B der Markierungsalgebra $\mathfrak{A}(A)$:

$$\alpha_{\mathcal{F}} : B \times B \to \mathbb{B} \tag{3.27}$$

Für einen Kantentyp $\mathcal{F}$ definieren wir ein Aktivierungsprädikat $\alpha_{\mathcal{F}}$ mit $m, i \in \mathfrak{A}(A)$ wie folgt, wobei m die Markierung der Stelle und i das aus der Kanteninschrift überführte Element der Markierungsstruktur $\mathfrak{A}$ über der Markenmenge A sei.

$$\text{für alle } \mathcal{F} \in \{\mathcal{F}_{\text{pO}}\} \cup \Phi_{\text{i}} \text{ sei } \alpha_{\mathcal{F}}(m,i) = \begin{cases} \text{wahr} & \text{wenn } m \oplus i \text{ ist definiert} \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.28a}$$

$$\text{für alle } \mathcal{F} \in \{\mathcal{F}_{\text{RD}}\} \cup \Phi_{\text{d}} \text{ sei } \alpha_{\mathcal{F}}(m,i) = \begin{cases} \text{wahr} & \text{wenn } m \ominus i \text{ ist definiert} \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.28b}$$

$$\alpha_{\mathcal{F}_{\text{E}}}(m,i) = \begin{cases} \text{wahr} & \text{wenn } m = i \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.28c}$$

$$\alpha_{\mathcal{F}_{\text{Z}}}(m,i) = \begin{cases} \text{wahr} & \text{wenn } m = \varepsilon \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.28d}$$

$$\alpha_{\mathcal{F}_{\text{PI}}}(m,i) = \begin{cases} \text{wahr} & \text{wenn } m \oplus i \text{ ist nicht def.} \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.28e}$$

$$\alpha_{\mathcal{F}_{\text{I}}}(m,i) = \begin{cases} \text{wahr} & \text{wenn } m \ominus i \text{ ist nicht def.} \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.28f}$$

$$\text{für alle } \mathcal{F} \in \Phi_{\text{w}} \text{ sei } \alpha_{\mathcal{F}}(m,i) = \text{wahr} \tag{3.28g}$$

3.4.3 Effekt

Für alle Kantentypen, deren Klassifikationstyp nicht *read* ist, ist ein Effekt auf die Stelle der Kante definiert (vgl. Tab. 3.1). Er bezieht sich (wie die Aktivierungsbedingung der Kante) auf die Markierung der Stelle und die Kanteninschrift, die in ein Element derselben Algebra überführt wird, wie die Markierung der Stelle.

Die Effektfunktion $\eta_{\mathcal{F}}$ eines Kantentyps $\mathcal{F}$ definieren wir für Kantentypen $\mathcal{F} \in \Phi$ über den Elementen der Grundmenge B der Markierungsalgebra $\mathfrak{A}(A)$, wobei $\mathfrak{A}$ die Markierungsstruktur und A die Markenmenge eines Petrinetz-Typs sei.

$$\eta_{\mathcal{F}} : B \times B \to B \tag{3.29}$$

Für einen Kantentyp $\mathcal{F}$ definieren wir jeweils eine Effektfunktion $\eta_{\mathcal{F}}$ mit $m, i \in \mathfrak{A}(A)$ wie folgt, wobei m die Markierung der Stelle und i das aus der Kanteninschrift überführte Element der Markierungsstruktur $\mathfrak{A}$ über der Markenmenge A sei.

$$\text{für alle } \mathcal{F} \in \Phi_{\mathrm{d}} \text{ sei } \eta_{\mathcal{F}}(m, i) = m \ominus i \tag{3.30a}$$
$$\text{für alle } \mathcal{F} \in \Phi_{\mathrm{i}} \text{ sei } \eta_{\mathcal{F}}(m, i) = m \oplus i \tag{3.30b}$$
$$\eta_{\mathcal{F}_{\mathrm{S}}}(m, i) = i \tag{3.30c}$$
$$\eta_{\mathcal{F}_{\mathrm{C}}}(m, i) = \varepsilon \tag{3.30d}$$
$$\eta_{\mathcal{F}_{\mathrm{RF}}}(m, i) = m \tag{3.30e}$$

Da die Kantentypen der Menge Φ_{r} explizit keinen Effekt auf die Stelle der entsprechenden Kante haben, definieren wir auch keine Effektfunktion für diese Kantentypen.

3.5 Parametrisierte Schaltregel

Eine Klassifikation von Petrinetz-Typen ist nur sinnvoll, wenn damit möglichst viele Petrinetz-Typen erfasst werden. Einige Petrinetz-Typen können jedoch mit dem in Abschn. 3.2 vorgestellten Petrinetz-Würfel nicht beschrieben werden, da dafür die bisherige Definition der Schaltregel (Def. 3.2.2) nicht ausreicht. Ziel dieses Abschnittes ist eine Schaltregel, die über die Belegung von Parametern an klar definierten Stellen in ihrer Ausprägung verändert werden kann. Eine solche parametrisierte Schaltregel bildet das Grundmuster für alle Petrinetz-Schaltregeln.

Aus Abschn. 3.2.3 kennen wir Petrinetz-Typen als Paar $(A, \mathfrak{A})$. Wir werden später in Abschn. 3.6 darauf aufbauend Petrinetz-Typen des Petrinetz-Hyperwürfels definieren. Hier wird zunächst mit der Markenmenge A und der Markierungsstruktur $\mathfrak{A}$ eines Petrinetz-Typs gearbeitet.

Der Ausgangspunkt der Darstellungen in diesem Abschnitt ist der folgende beobachtende Vergleich von Schaltregeln verschiedener Petrinetz-Typen. Für jede Transition eines Petrinetzes kann bestimmt werden, ob sie unter der aktuellen Markierung des Netzes konzessioniert ist (Abschn. 3.5.1). Mehrere Transitionen werden in einigen Petrinetz-Typen zu einem gemeinsam schaltenden Schritt zusammengefasst. Einzeln schaltende Transitionen passen zu dieser Sichtweise, indem sie Einzelschritte bilden. Wir unterscheiden eine lokale Bestimmung, ob eine Ansammlung von Transitionen einen Schritt bildet (Abschn. 3.5.2), von einer globalen Betrachtung aller Schritte

und einer Auswahl eines tatsächlich aktivierten Schrittes (Abschn. 3.5.3). Letztendlich wird mit der parametrisierten Schaltregel bestimmt, welcher aktivierte Schritt in einem Petrinetz schalten kann.

3.5.1 Lokale Aktivierung – Belegung

Die Markierung einer Stelle ist ein Element der Algebra, die wir als Markierungsstruktur bezeichnet haben, mit den Marken als atomaren Elementen. Kanteninschriften werden in einem Schaltmodus ebenfalls zu Elementen der Algebra ausgewertet (vgl. Abschn. 3.2.3), damit Marken von Stellen entfernt bzw. zu Stellen hinzugefügt werden können. Das heißt, es gibt eine Folge von Abbildungen (hier mit j bezeichnet), die einer Kante eines Netzes N eine Kanteninschrift zuordnet, die letztlich in ein Element der Markierungsstruktur ($\mathfrak{A}(A)$) des Petrinetz-Typs von N ausgewertet wird:

$$\begin{aligned} j_0 &: F_N \to X_1 \\ j_1 &: X_1 \to X_2 \\ &\vdots \\ j_n &: X_n \to \mathfrak{A}(A) \end{aligned} \tag{3.31}$$

Beispielsweise sieht für S/T-Netze diese Folge von Abbildungen üblicherweise so aus

$$\begin{aligned} j_0 &: F_N \to \mathbb{N} \\ j_1 &: \mathbb{N} \to \mathfrak{M}_\mathfrak{s}(\{\bullet\}) \end{aligned} \tag{3.32}$$

Ein Kanteninschrift ist hier eine natürliche Zahl, die in eine Multimenge von schwarzen Marken überführt wird.

Das am häufigsten verwendete Verfahren, eine Kanteninschrift in einem Schaltmodus eines *high level* Netzes[3] auszuwerten, ist über die Belegung von freien Variablen. Sei $\mathcal{X}$ die Menge der freien Variablen, dann ist die Folge j von Abbildungen definiert durch:

$$\begin{aligned} j_0 &: F_N \to X_1 \\ j_1 &: X_1 \to X_2 \\ &\vdots \\ j_n &: \mathfrak{A}(A \cup \mathcal{X}) \to \mathfrak{A}(A) \end{aligned} \tag{3.33}$$

Im Folgenden beschränken wir uns auf derartige Folgen von Abbildungen, für den allgemeinen (weniger relevanten) Fall gilt entsprechendes.

[3] *Low level* Netze haben auf Grund der einelementigen Markenmenge keine verschiedenen Schaltmodi.

Wir bezeichnen die (letzte) Abbildung j_n in einer Folge von Abbildungen wie in (3.33) als Belegung δ. Dabei werden Ausdrücke i verwendet, die aus Elementen der Markenmenge A und freien Variablen ($\mathcal{X}$) zusammengesetzt sind und so Elemente einer Markierungsalgebra $\mathfrak{A}(A \cup \mathcal{X})$ bilden. Jede freie Variable aus $\mathcal{X}$ ist also ebenfalls ein atomares Element bezüglich der Markierungsalgebra $\mathfrak{A}(A \cup \mathcal{X})$. Der Ausdruck i wird ausgewertet, indem jeder freien Variablen ein Element der Markenmenge A zugeordnet wird. Gleiche Variable erhalten dabei gleiche Werte. Wir bezeichnen die Ersetzung einer freien Variablen $x \in \mathcal{X}$ (d. h. jedes Vorkommen von x) in einem Ausdruck i durch ein Element $a \in A$ mit $i_{|x:a}$

Definition 3.5.1 (Belegung). *Sei A eine Menge von Marken, $\mathfrak{A}$ eine Markierungsstruktur und $\mathcal{X}$ eine Menge freier Variablen mit $A \cap \mathcal{X} = \emptyset$, dann heißt die Abbildung $\delta : \mathfrak{A}(A \cup \mathcal{X}) \to \mathfrak{A}(A)$* Belegung, *wenn für alle $m \in \mathfrak{A}(A)$ gilt $\delta(m) = m$ und wenn es eine Abbildung $\overline{\delta} : \mathcal{X} \to A$ so gibt, dass für alle $i \in \mathfrak{A}(A \cup \mathcal{X})$ und für alle $x_1, \ldots, x_n \in \mathcal{X}$ gilt $\delta(i) = \delta(i_{|x_1:\overline{\delta}(x_1)}) = \delta((i_{|x_1:\overline{\delta}(x_1)})_{|x_2:\overline{\delta}(x_2)}) = \cdots = \delta((\cdots(i_{|x_1:\overline{\delta}(x_1)})\cdots)_{|x_n:\overline{\delta}(x_n)})$ und $\delta((\cdots(i_{|x_1:\overline{\delta}(x_1)})\cdots)_{|x_n:\overline{\delta}(x_n)}) \in \mathfrak{A}(A)$.* ∗

Im Allgemeinen gibt es für einen Ausdruck i beliebig viele Belegungen, da jede in i vorkommende Variable durch ein Element der beliebig großen (endlichen) Markenmenge A belegt werden kann. Eine Belegung wird mit einer Transition zu einem Schaltmodus zusammengefasst.

Definition 3.5.2 (Schaltmodus). *Ein* Schaltmodus *ist ein Paar (t, δ) aus einer Transition t und einer Belegung δ. Dabei ist t die* Transition des Schaltmodus *und δ* seine Belegung. ∗

Jedes Label $L : Y \to X$ hat in seinem Universum ein Aktivierungsprädikat $v : X \to (\Theta \to \mathbb{B})$ (siehe Abschn. 3.3), das einem Schaltmodus $(t, \delta) \in \Theta$ einen Wahrheitswert zuweist. Das Prädikat $v(L(y))(t, \delta)$ mit $y \in Y$ ist wahr, wenn das Label L des entsprechenden Netzelements y mit dem Schaltmodus (t, δ) gültig ist.

In trivialen Fällen hat ein Label das Aktivierungsprädikat v_{def} (siehe Abschn. 3.3), das für jeden Wert und jede Belegung den Wahrheitswert wahr liefert. Der Name eines Knotens hat z. B. ein derartiges triviales Aktivierungsprädikat. Für die Label, die in der Schaltregel eine Bedeutung haben, wird das Aktivierungsprädikat in seinem Universum festgelegt. So ist z. B. eine Zeitanschrift an Stellen eines Netzes N, die die Mindestaufenthaltsdauer der Marken auf der Stelle modelliert [Sta95] so definiert:

$$L_{\text{time-dur}} : P_N \to \mathbb{N} \tag{3.34}$$

Für das Aktivierungsprädikat v gilt dann beispielsweise mit einer Uhr je Stelle ($L_{\text{clock}} : P_N \to \mathbb{N}$)

$$v(L_{\text{time-dur}}(p))(t, \delta) = \begin{cases} \text{wahr} & \text{wenn } L_{\text{time-dur}}(p) \leq L_{\text{clock}}(p) \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.35}$$

Für einen *transition guard* (zusätzliche Aktivierungsbedingung der Transition) gilt beispielsweise

$$L_{\text{guard}} : T \to (O \times \mathfrak{A}(A \cup \mathcal{X}) \times \cdots \times \mathfrak{A}(A \cup \mathcal{X})) \tag{3.36}$$

wobei jedes $o \in O$ eine Abbildung $o : \mathfrak{A}(A) \times \cdots \times \mathfrak{A}(A) \to \mathbb{B}$ ist. Dann gilt für das Aktivierungsprädikat v eines *transition guard*

$$v(L_{\text{guard}}(t))(t, \delta) = L_{\text{guard}}(t)_{(1)}(\delta(L_{\text{guard}}(t)_{(2)}), \ldots, \delta(L_{\text{guard}}(t)_{(n)})) \tag{3.37}$$

Jeder Kante eines Petrinetzes N ist, wie in Abschn. 3.3 ausgeführt, (wenn auch implizit) eine Kanteninschrift zugeordnet. Wie alle anderen Label ist auch die Kanteninschrift eine Abbildung in eine beliebige Menge (vgl. Gl. (3.33)); also

$$i' : F \to Y_i \tag{3.38}$$

Diese Menge Y_i bezeichnen wir als Menge von *Markierungstermen*, denn die Kanteninschrift beschreibt eine Markierung (in Abhängigkeit vom Schaltmodus ihrer Transition), die mit der Markierung auf der Stelle der Kante „verrechnet“ wird.

Wir müssen nun dafür sorgen, Markierungsterme so in Elemente einer Struktur von Markentermen zu überführen, dass sie zur Markierungsstruktur „passen“ und von einer Belegung „erfasst“ werden. Wir definieren eine Funktion i'' zur Überführung einer Kanteninschrift in eine Struktur von Markentermen

$$i'' : Y_i \to \mathfrak{A}(A \cup \mathcal{X}) \tag{3.39}$$

Die Struktur von Markentermen ist eine Algebra passend zur Markierungsstruktur des Netztyps mit zusätzlichen Elementen, die mit Hilfe freier Variablen aus $\mathcal{X}$ gebildet werden.

Als Beispiel betrachten wir die Funktion i'' für S/T-Netze. Üblicherweise verwendet man natürliche Zahlen als Kanteninschriften in einem S/T-Netz N ($i' : F_N \to \mathbb{N}$), und Variablen sind in S/T-Netzen nicht erlaubt ($\mathcal{X} = \emptyset$). Damit gilt für i'' und $n \in \mathbb{N}$, $n > 0$:

$$i'' : \mathbb{N} \to \mathfrak{M}\mathfrak{s}(\{\bullet\} \cup \emptyset) \tag{3.40}$$

$$\text{also} \quad i'' : \mathbb{N} \to \mathcal{B}(\{\bullet\}) \quad \text{, da für } \mathfrak{M}\mathfrak{s}(A) \text{ gilt } B = \mathcal{B}(A) \tag{3.41}$$

$$i''(1) = [\bullet] \tag{3.42}$$

$$i''(n+1) = i''(n) \oplus [\bullet] \tag{3.43}$$

Im Folgenden fassen wir i' (3.38) und i'' (3.39) mit i zusammen. Für $i : F_N \to \mathfrak{A}(A \cup \mathcal{X})$ sei $i(f) = i''(i'(f))$.

Mit der in diesem Abschnitt eingeführten Belegung und der aus Abschn. 3.4.2 bekannten Aktivierungsbedingung für Kanten definieren wir, wann ein *Kante* aktiviert ist.

Definition 3.5.3 (Aktivierung einer Kante). *Die Kante $f \in F_N$ eines mit M markierten Netzes N ist mit der Kanteninschrift $i(f)$ und dem Schaltmodus (f_T, δ) aktiviert, wenn gilt $\alpha_{\varphi(f)}(M(f_\mathrm{P}), \delta(i(f)))$.* *

Darauf aufbauend betrachten wir nun, wie in Abschn. 3.4.1 am Beispiel gezeigt (vgl. Abb. 3.9), wann eine Menge von Kanten zwischen einer Stelle und einer Transition aktiviert ist. Dabei muss zunächst jede Kante aktiviert sein. Zum zweiten dürfen sich die Kanten nicht widersprechen, d. h. es darf neben einer *write*-Kante keine weitere *write*-Kante und keine *delete*-Kante geben. Und zum dritten muss der kumulierte Effekt auf der Stelle möglich sein. Bei der Berechnung des kumulierten Effektes einer Kantenmenge auf eine Stelle wird zunächst der Effekt aller *delete*-Kanten bzw. der einen *write*-Kante ermittelt, woraufhin der Effekt aller *insert*-Kanten hinzugefügt wird. Ausgangspunkt dieser Definition ist, dass in Petrinetzen nur Marken fließen (konsumiert werden), die bereits vorhanden sind und nicht erst ein Ergebnis des Schaltens sind. Deshalb werden erst Marken konsumiert (*delete*) bzw. gesetzt (*write*) und dann Marken hinzugefügt (*insert*). Wir verfolgen damit den Ansatz des *rewriting* und lassen ein *safe delete* [DR98] an dieser Stelle außer Acht. Da eine *read*-Kante explizit keinen Effekt auf ihre Stelle habt, muss sie auch nicht für einen kumulierten Effekt herangezogen werden. Ihre Aktivierungsbedingung geht bei der Überprüfung der Aktivierung einer Kante (Def. 3.5.3) ein.

Definition 3.5.4 (Aktivierung einer Kantenmenge). *Sei N ein Netz und M seine Markierung. Die Kantenmenge $F_{p,t} \subseteq F_N$ zwischen einer Stelle p und einer Transition t (für alle $f, f' \in F_{p,t}$ sei $R_N(f) = R_N(f')$) ist bei $M(p)$ für einen Schaltmodus (t, δ) aktiviert, wenn*

1. *jede Kante mit ihrer Kanteninschrift $i(f)$ und der Belegung (t, δ) aktiviert ist (siehe Definition 3.5.3), d. h. für alle $f \in F_{p,t}$ gilt $\alpha_{\varphi(f)}(M(f_\mathrm{P}), \delta(i(f)))$*
2. *es eine* write*-Kante gibt, dann gibt es keine weitere* write*-Kante und keine* delete*-Kante, d. h. wenn es ein $f \in F_{p,t}$ mit $\varphi(f) \in \Phi_\mathrm{w}$ gibt, dann gibt es kein $f' \in F_{p,t}$ mit $f' \neq f$ und $\varphi(f') \in \Phi_\mathrm{w} \cup \Phi_\mathrm{d}$,*
3. *der Effekt aller* delete*-Kanten bzw. der* write*-Kante auf der Stelle p zulässig ist, d. h. mit $\{f_1, \ldots, f_n\} = \{f \mid f \in F_{p,t} \text{ und } \varphi(f) \in \Phi_\mathrm{d}\}$ gibt es ein $m_1 \in \mathfrak{A}(A)$ (m_1 ist definiert) mit:*

$$m_1 = \begin{cases} \eta_{\varphi(f_n)}(\ldots(\eta_{\varphi(f_2)}(\eta_{\varphi(f_1)}(M(p), \delta(i(f_1))), \\ \qquad\qquad \delta(i(f_2))), \ldots), \\ \qquad \delta(i(f_n))) & \text{wenn es ein } f \in F_{p,t} \\ & \text{mit } \varphi(f) \in \Phi_\mathrm{d} \text{ gibt} \\ \eta_{\varphi(f)}(M(p), \delta(i(f))) & \text{wenn es ein } \exists f \in F_{p,t} \\ & \text{mit } \varphi(f) \in \Phi_\mathrm{w} \text{ gibt} \\ M(p) & \text{sonst} \end{cases}$$

4. *der Effekt aller* insert*-Kanten auf m_1 zulässig ist, d. h. mit $\{f_1, \ldots, f_n\} = \{f \mid f \in F_{p,t}$ und $\varphi(f) \in \Phi_\mathrm{i}\}$ gibt es ein $m_2 \in \mathfrak{A}(A)$ (m_2 ist definiert) mit:*

$$m_2 = \begin{cases} \eta_{\varphi(f_n)}(\ldots(\eta_{\varphi(f_2)}(\eta_{\varphi(f_1)}(m_1, \delta(i(f_1))), \\ \qquad\qquad \delta(i(f_2))), \ldots), \\ \qquad \delta(i(f_n))) & \text{wenn es ein } f \in F_{p,t} \\ & \text{mit } \varphi(f) \in \Phi_\mathrm{i} \text{ gibt} \\ m_1 & \text{sonst} \end{cases}$$

*

Für Markierungsstrukturen, die eine Reihenfolge ihrer Elemente annehmen (z. B. Sequenzen und Stapel) ergibt sich mit dieser Definition ein Nichtdeterminismus über der Reihenfolge in der der Effekt der Kanten berechnet wird. Das heißt, m_1 und m_2 in Def. 3.5.4 sind für derartige Markierungsstrukturen nicht eindeutig bestimmt. Ohne Beschränkung der Allgemeinheit abstrahieren wir von diesem Nichtdeterminismus, denn er ist nur dann ein Problem[4], wenn es mehrere Kanten mit gleichem Klassifikationstyp zwischen einer Stelle und einer Transition gibt bzw. in Netzen mit Schrittschaltregel.

Im Gegensatz zu *low level* Netzen, in denen wir von konzessionierten Transitionen sprechen, sprechen wir hier von konzessionierten Schaltmodi. Ein konzessionierter Schaltmodus ist ein Paar einer Transition und einer Belegung so, dass die Transition beim Schalten auf allen Stellen ihrer Umgebung korrekte Markierungen erzeugt. Das heißt die jeweilige Menge der Kanten zwischen der Transition und jeder Stelle ihrer Umgebung ist mit der Belegung des Schaltmodus aktiviert nach Def. 3.5.4.

Definition 3.5.5 (konzessionierter Schaltmodus). *Ein Schaltmodus (t, δ) heißt* konzessioniert*, wenn die Menge aller adjazenten Kanten von t mit der gleichen Stelle bezogen je auf die entsprechende Stelle und die Transition t aktiviert sind.* *

Ein konzessionierter Schaltmodus ist nicht aktiviert, wenn das Aktivierungsprädikat eines Labels im Umfeld der Transition des Schaltmodus nicht aktiviert ist. Wir definieren deshalb einen aktivierten Schaltmodus.

Definition 3.5.6 (aktivierter Schaltmodus). *Ein konzessionierter Schaltmodus (t, δ) heißt* aktiviert*, wenn die Aktivierungsprädikate v aller Label L der Netzelemente im Umfeld der Transition t mit der Belegung δ gültig sind. Sei $x \in \dot{t}$, dann gilt $v(L(x))(t, \delta)$.* *

[4] Das Problem besteht darin, dass keine eindeutige Nachfolgemarkierung einer Stelle beim Schalten einer entsprechenden Transition berechnet werden kann. Das hat Auswirkungen auf die Erreichbarkeitsanalyse u. ä.

Wie oben bereits ausgeführt, gibt es für eine Transition eines *low level* Netzes immer genau eine Belegung. Damit ist der Begriff des aktivierten Schaltmodus für *low level* Netze gleich bedeutend mit dem Begriff der aktivierten Transition, wie er bereits in Abschn. 2.2 verwendet wurde.

Im Folgenden sei Θ die Menge aller Schaltmodi eines Netzes N mit der Markierung M, für das jedes $(t, \delta) \in \Theta$ konzessioniert ist.

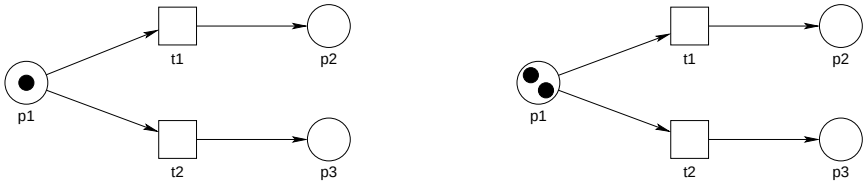
3.5.2 Schritte

In Abschn. 3.2.3 haben wir das Schalten lediglich einer Transition in einem Schritt betrachtet. Häufig (z. B. in Signal-Netzsystemen) werden Petrinetze jedoch unter einer Schrittschaltregel betrachtet, die einige Probleme mit sich bringt. Beispielsweise muss durch die Schaltregel sichergestellt werden, dass ein und dieselbe Marke nicht von verschiedenen Transitionen eines Schrittes konsumiert wird. Dies widerspräche unserer Grundannahme von der Atomarität der Marken.

Mit Hilfe der Netze in Abb. 3.11 wird dies erläutert. Die Transitionen t1 und t2 in den Abbn. 3.11(a) und 3.11(b) haben jeweils Konzession. In dem Netz aus Abb. 3.11(a) besteht ein Schritt aus jeweils einer Transition, da auf p1 nur eine Marke liegt. Die beiden Transitionen können also nicht gemeinsam in einem Schritt schalten. In dem Netz aus Abb. 3.11(b) können t1 und t2 auch gemeinsam einen Schritt bilden, da der gemeinsame (kumulierte) Effekt ihres Schaltens auf p1 möglich ist.

Das Netz in Abb. 3.12 zeigt ein S/T-Netz mit einer READ-Kante (siehe Tab. 3.1) zwischen p2 und t1 vor und nach dem Schalten des Schrittes bestehend aus den beiden Transitionen t1 und t2. Die beiden Transitionen (Abb. 3.12(a)) sind gemeinsam in einem Schritt aktiviert, da t1 beim Schalten die Marke auf p2 nicht konsumiert sondern lediglich als Aktivierungsbedingung benötigt.

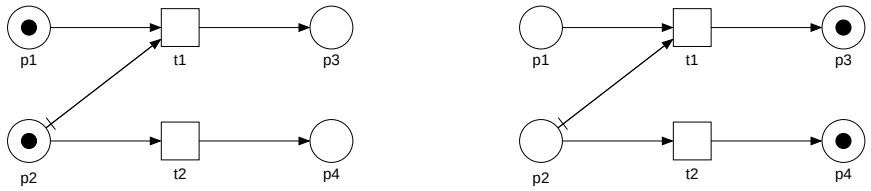
Im Folgenden führen wir den ersten Parameter der parametrisierten Schaltregel ein. Dieser Parameter bestimmt die Zusammensetzung eines Schrittes und wird mit ξ bezeichnet. Der Parameter wird auf eine Menge von Schaltmodi angewendet, heißt *Schrittparameter* und liefert die Menge aller Multimengen von Schaltmodi, die sich mit



(a) 2 einelementige Schritte möglich

(b) 3 Schritte möglich

Abbildung 3.11: Schrittschaltregel



(a) Schritt (t1,t2) aktiviert (b) Schritt (t1,t2) geschaltet

Abbildung 3.12: Schrittschaltregel in S/T-Netz mit READ-Kante

diesem Parameter bilden lassen. Wir unterscheiden die folgenden konkreten Parameter mit den angegebenen Definitionen.

$$\xi_{\mathrm{Sg}}(X) = \{[x] \mid x \in X\} \tag{3.44}$$

$$\xi_{\mathrm{St}}(X) = \{Y \mid Y \in \mathcal{B}(X) \text{ und für alle } y \text{ mit } Y(y) \geq 1 \text{ gilt } Y(y) = 1\} \tag{3.45}$$

$$\xi_{\mathrm{Ms}}(X) = \{Y \mid Y \in \mathcal{B}(X)\} \tag{3.46}$$

$\xi_{\mathrm{Sg}}(X)$ beschreibt alle Einermengen über einer Menge X, $\xi_{\mathrm{St}}(X)$ alle Teilmengen und $\xi_{\mathrm{Ms}}(X)$ alle Multimengen über X jeweils als eine Menge von Multimengen. Damit ist $\Theta^+ \in \xi(\Theta)$ eine Multimenge von Schaltmodi.

Wir benötigen später die Menge der Transitionen, die in einer Multimenge von Schaltmodi enthalten sind. Also definieren wir

$$T_{\Theta^+} = \{t \mid \text{es gibt ein } (t, \delta) \in \Theta^+\} \tag{3.47}$$

Wir definieren den kumulierten Effekt $\eta^+_{\mathcal{F}}$ einer Menge von Kanten mit gleichem Typ $\mathcal{F}$ induktiv. Seien $m_1, m_2 \in \mathfrak{A}(A)$ und $n \in \mathbb{N}$, $n \geq 1$. Dann sei der kumulierte Effekt einer Menge von Kanten gleichen Typs (vgl. Abschn. 3.4.3)

$$\eta^+_{\mathcal{F}}(m_1, m_2, n) = \begin{cases} \eta_{\mathcal{F}}(m_1, m_2) & \text{wenn } n = 1 \\ \eta^+_{\mathcal{F}}(\eta_{\mathcal{F}}(m_1, m_2), m_2, n-1) & \text{wenn } n > 1 \end{cases} \tag{3.48}$$

Wir benötigen den kumulierten Effekt einer Menge von Kanten, um den kumulierten Effekt auf einer Stelle zu berechnen.

Zu einer Multimenge von Schaltmodi Θ^+ definieren wir die Multimenge von Paaren F^+_Δ aus einer Kante f und einer Belegung δ. Die Transition der Kante und die Belegung sind als Schaltmodus (f_{T}, δ) genauso oft in der Multimenge der Schaltmodi enthalten, wie das Paar (f, δ) in der Multimenge F^+_Δ. Wir definieren die Multimenge F^+_Δ induktiv über alle $f \in F_N$ und alle $(f_{\mathrm{T}}, \delta) \in \Theta^+$.

$$F^+_\Delta(f, \delta) = \begin{cases} \Theta^+(t, \delta) & \text{wenn } t = f_{\mathrm{T}} \text{ und } (t, \delta) \in \Theta^+ \\ 0 & \text{sonst} \end{cases} \tag{3.49}$$

Wir definieren die Menge aller Stellen P_{F^+}, die Stellen von Kanten aus den Paaren $(f, \delta) \in F^+_\Delta$ sind; für alle $f \in f_N$ sei

$$P_{F^+} = \{f_\mathrm{P} \mid \text{es gibt ein } (f, \delta) \in F^+_\Delta\} \tag{3.50}$$

Der kumulierte Effekt auf einer Stelle ist nur dann zulässig, wenn auch die Label der Stelle, den Effekt für zulässig halten. Jedes Label L einer Stelle eines Netzes N mit der Markenmenge A und der Markierungsstruktur $\mathfrak{A}$, das heißt, die Label mit den Labeluniversen $\mathfrak{l} \in \mathfrak{L}$ eines Petrinetz-Typs mit $\mathfrak{l}_{(1)} = P_N$ ($L : \mathfrak{l}_{(1)} \to \mathfrak{l}_{(2)}$ mit $\mathfrak{l}_{(1)} = P_N$) hat ein Effektprädikat k, das die Zulässigkeit des kumulierten Effektes auf eine Stelle $p \in P_N$ einschränkt. Es bezieht sich auf die Elemente der Markierungsstruktur der Stelle $k : L \to (\mathfrak{A}(A) \to \mathbb{B})$. In trivialen Fällen ist k für ein Label L der Stelle p für jedes Element der Markierungsstruktur wahr ($k(L(p))(m) =$ wahr für alle $m \in \mathfrak{A}(A)$).

Wir stellen nun den Kern dieses Abschnittes dar. Eine Multimenge von Schaltmodi ist dann aktiviert, wenn wenigstens der Effekt auf den Stellen der Umgebung aller Transitionen der Schaltmodi zulässig ist. Ähnlich wie schon für den Effekt einer Kantenmenge zwischen einer Transition und einer Stelle, können wir den Effekt aller Kanten, die mit einer Stelle verbunden sind, mit allen in Frage kommenden Schaltmodi ermitteln. Auf eine Stelle dürfen beliebig viele *read*-Kanten in einem Schritt zugreifen. In einem Schritt darf auf eine Stelle maximal eine *write*-Kante zugreifen, da diese Kante die gesamte Markierung der Stelle verändert. Wenn es eine *write*-Kante gibt, darf es keine *delete*-Kante geben. Wenn es keine *write*-Kante gibt, dürfen beliebig viele *delete*-Kanten auf die Stelle zugreifen, wenn ihr kumulierter Effekt zulässig ist, das heißt, wenn das Ergebnis der Kombination ihrer Operationen in der Algebra der Stelle definiert ist. Letzteres wird auch für die *insert*-Kanten der Stelle gefordert; sie sind in einem Schritt zulässig, wenn ihr kumulierter Effekt in der Algebra der Stelle definiert ist.

Definition 3.5.7 (zulässiger kumulierter Effekt). *Für eine Stelle $p \in P_{F^+}$ (P_{F^+} wie oben), eine Markierung M eines Netzes N und eine Multimenge von Schaltmodi Θ^+ heißt der kumulierte Effekt zulässig, wenn*

1. *es eine write-Kante gibt, die auf die Stelle zugreift, dann gibt es keine weitere write-Kante und keine delete-Kante, d. h. wenn es ein $f \in F_N$ mit $f_\mathrm{P} = p$ und ein $(f, \delta) \in F^+_\Delta$ und $\varphi(f) \in \Phi_\mathrm{w}$ gibt, dann gilt*
   a) *$F^+_\Delta(f, \delta) = 1$ und*
   b) *es gibt kein $(f, \delta') \in F^+_\Delta$ mit $\delta' \neq \delta$ und*
   c) *es gibt kein $(f', \delta') \in F^+_\Delta$ mit $f' \neq f$ und $\delta' \neq \delta$ und $\varphi(f') \in \Phi_\mathrm{w} \cup \Phi_\mathrm{d}$*
2. *der kumulierte Effekt aller delete-Kanten bzw. der einen write-Kante auf p zulässig ist, d. h. mit $\{(f_1, \delta_1), \ldots, (f_n, \delta_n)\} = \{(f, \delta) \mid (f, \delta) \in F^+_\Delta$ und $f_\mathrm{P} =$*

p und $\varphi(f) \in \Phi_\mathrm{d}\}$ gibt es ein $M' \in \mathfrak{A}(A)$ (M' ist definiert) mit:

$$M'(p) = \begin{cases} \eta^+_{\varphi(f_n)}(\dots(\eta^+_{\varphi(f_2)}(\eta^+_{\varphi(f_1)}(M(p), & \\ \qquad \delta(i(f_1)), & \\ \qquad F^+_\Delta(f_1, \delta_1)), & \\ \qquad \delta(i(f_2)), & \\ \qquad F^+_\Delta(f_2, \delta_2)), \dots), & \\ \qquad \delta(i(f_n)), F^+_\Delta(f_n, \delta_n)) & \text{wenn es ein } (f,\delta) \in F^+_\Delta \text{ mit} \\ & f_\mathrm{P} = p \text{ und } \varphi(f) \in \Phi_\mathrm{d} \text{ gibt} \\ \eta_{\varphi(f)}(M(p), \delta(i(f))) & \text{wenn es ein } (f,\delta) \text{ mit} \\ & F^+_\Delta(f,\delta) = 1 \text{ und } f_\mathrm{P} = p \text{ und} \\ & \varphi(f) \in \Phi_\mathrm{w} \text{ gibt} \\ M(p) & \text{sonst} \end{cases}$$

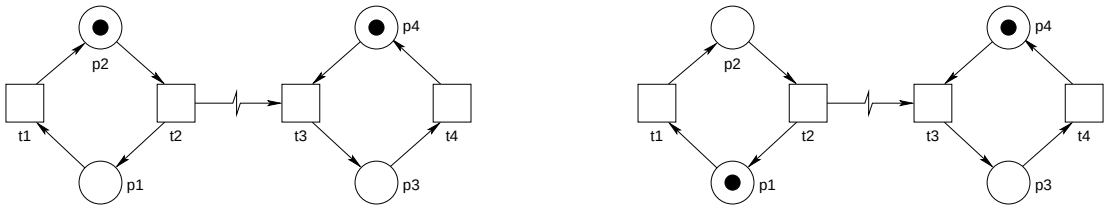
3. *der kumulierte Effekt aller insert-Kanten auf p zulässig ist, d. h. mit $\{(f_1,\delta_1),\dots,(f_n,\delta_n)\} = \{(f,\delta) \mid (f,\delta) \in F^+_\Delta$ und $f_\mathrm{P} = p$ und $\varphi(f) \in \Phi_\mathrm{i}\}$ gibt es ein $M'' \in \mathfrak{A}(A)$ (M'' ist definiert) mit:*

$$M''(p) = \begin{cases} \eta^+_{\varphi(f_n)}(\dots(\eta^+_{\varphi(f_2)}(\eta^+_{\varphi(f_1)}(M'(p), & \\ \qquad \delta(i(f_1)), & \\ \qquad F^+_\Delta(f_1, \delta_1)), & \\ \qquad \delta(i(f_2)), & \\ \qquad F^+_\Delta(f_2, \delta_2)), \dots), & \\ \qquad \delta(i(f_n)), F^+_\Delta(f_n, \delta_n)) & \text{wenn es ein } (f,\delta) \in F^+_\Delta \text{ mit} \\ & f_\mathrm{P} = p \text{ und } \varphi(f) \in \Phi_\mathrm{i} \text{ gibt} \\ M'(p) & \text{sonst} \end{cases}$$

4. *alle Effektprädikate von Labeln der Stelle gültig sind, d. h. wenn für alle Label L der Stelle gilt $k(L(p))(M''(p)) =$* wahr

Der zulässige kumulierte Effekt auf die Stelle p ist die Markierung $M''(p)$. $*$

Das Prädikat k bestimmter Label einer Stelle kontrolliert den zulässigen kumulierten Effekt. Es ist vom Petrinetz-Typ abhängig und wird bspw. benötigt, um Kapazitäten von Stellen auszudrücken. Im Sinne unseres Ansatzes des *rewriting* (vgl.



(a) Transition t3 kann im Schritt (t2,t3) schalten

(b) Transition t3 kann nicht schalten

Abbildung 3.13: Schrittschaltregel in Signal-Netzsystemen

S. 60) formulieren wir mit einem entsprechenden Effektprädikat eines Stellen-Labels die schwache Kapazität [DR98].

Mit Def. 3.5.7 abstrahieren wir, wie schon mit Def. 3.5.4 von dem Nichtdeterminismus in Markierungsalgebren, die eine nicht kommutative Operation $\oplus$ zulassen.

Wir können nun definieren, wann eine Multimenge von Schaltmodi aktiviert ist, nämlich dann, wenn alle kumulierten Effekte auf den Stellen in der Umgebung aller Transitionen der Schaltmodi, je entsprechend mit ihrer Belegung, zulässig sind.

Definition 3.5.8 (aktivierte Multimenge von Schaltmodi). *Eine Multimenge von Schaltmodi Θ^+ ist aktiviert, wenn jeder Schaltmodus $(t, \delta) \in \Theta^+$ konzessioniert ist und der kumulierte Effekt für alle Stellen in der Umgebung aller Transitionen der Multimenge von Schaltmodi Θ^+ zulässig ist.* *

Ein Schritt ist wenigstens eine aktivierte Multimenge von Schaltmodi. Mit der Def. 3.5.7 des zulässigen kumulierten Effektes haben wir ausgeschlossen, dass ein und dieselbe Marke mehrmals in einem Schritt konsumiert bzw. produziert wird. Lediglich ihre mehrfache Verwendung über *read*-Kanten ist aufgrund der Klassifikation von Kantentypen zulässig. Damit treffen wir den klassischen Konfliktbegriff in Petrinetzen (z. B. [Sta90, Jen92]). Für einige Petrinetze reicht der Ausschluss eines Konfliktes (um eine Marke) nicht aus, sondern es müssen weitere Bedingungen erfüllt sein, damit zwei Schaltmodi gemeinsam in einem Schritt schalten dürfen.

Als Beispiel betrachten wir die Abb. 3.13 eines Signal-Netzsystems. Signal-Netzsysteme [HL00] sind so definiert, dass eine Teilmenge der Schaltmodi einen Schritt bildet. Signalkanten forcieren das Schalten einer Transition, das heißt, ein Schaltmodus mit einer Transition t ist dann in einem Schritt enthalten, wenn sie spontan ist oder wenn alle Transitionen im Schritt enthalten sind, die ein Ereignissignal zu t senden. Zur Transition t3 in Abb. 3.13(a) finden wir einen konzessionierten Schaltmodus und gemeinsam mit t2 auch eine Multimenge von Schaltmodi, die einen Schritt bildet.

Im Gegensatz dazu finden wir zur Transition t3 in Abb. 3.13(b) zwar einen konzessionierten Schaltmodus, aber keinen Schritt, da es zur Transition t2 keinen konzessionier-

ten Schaltmodus gibt und damit auch keine aktivierte Multimenge von Schaltmodi, in dem t2 enthalten wäre. Die Transition t3 kann also nicht schalten.

Wir führen dafür den zweiten Parameter der Schaltregel ein, den wir mit ζ bezeichnen. Dieser Parameter ist ein Prädikat, das bestimmt, ob eine aktivierte Multimenge von Schaltmodi ein Schritt ist. Wir nennen diesen Parameter *Abhängigkeitsparameter*.

Im Allgemeinen werden Petrinetztypen den Abhängigkeitsparameter ζ so definieren, dass jede aktivierte Multimenge von Schaltmodi Θ^+ eines Netzes N mit einer Markierung M ein Schritt ist. Wir bezeichnen ein solches Prädikat mit ζ_{def} und definieren

$$\zeta_{\text{def}}(\Theta^+, (t, \delta)) = \text{wahr} \tag{3.51}$$

Dabei ist der Schaltmodus (t, δ) ein Kandidat, die Multimenge Θ^+ zu erweitern.

Es gibt jedoch Petrinetz-Typen, die für ζ eine andere Definition erfordern, so dass nicht jede aktivierte Multimenge von Schaltmodi auch ein Schritt ist. Beispielsweise legen Signal-Netzsysteme, wie oben im Beispiel zu sehen war, fest, dass nur solche Schaltmodi zu einem Schritt gehören, deren Transition eine spontane Transition ist oder die alle Ereignissignale empfängt. Eine Transition empfängt ein Ereignissignal, wenn die Signal sendende Transition im selben Schritt enthalten ist, wie die empfangende Transition. Wir modellieren den Empfang von Ereignissignalen durch ein Label $L_{\text{sig}} : T \to \mathcal{P}(T)$, wobei $L_{\text{sig}}(t)$ die Menge von Transitionen bezeichnet, von denen t ein Ereignissignal erwartet. Damit ist der Abhängigkeitsparameter ζ_{SN} für Signal-Netzsysteme wie folgt definiert

$$\zeta_{\text{SN}}(\Theta^+, (t, \delta)) = \begin{cases} \text{wahr} & \text{wenn } L_{\text{sig}}(t) = \emptyset \\ \text{wahr} & \text{wenn für alle } t' \in L_{\text{sig}}(t) \text{ gibt es ein } (t', \delta) \in \Theta^+ \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.52}$$

Ein anderes Beispiel ist der Ausschluss von Selbstnebenläufigkeit. Das heißt, ein Schaltmodus (t, δ) darf nicht in einem Schritt enthalten sein, wenn es einen Schaltmodus (t, δ') im Schritt gibt, der dieselbe Transition t enthält.

$$\zeta_{\text{self}}(\Theta^+, (t, \delta)) = \begin{cases} \text{falsch} & \text{wenn } \Theta^+(t, \delta) > 1 \\ \text{falsch} & \text{wenn es ein } (t', \delta') \in \Theta^+ \text{ mit } \delta' \neq \delta \text{ und } t' = t \text{ gibt} \\ \text{wahr} & \text{sonst} \end{cases} \tag{3.53}$$

Mit dem Abhängigkeitsparameter ζ definieren wir einen Schritt.

Definition 3.5.9 (Schritt). *Eine aktivierte Multimenge von Schaltmodi Θ^+ ist ein* Schritt *von (N, M), wenn für alle Schaltmodi $(t, \delta) \in \Theta^+$ gilt, dass das globale Prädikat $\zeta(\Theta^+ \boxminus [(t, \delta)], (t, \delta))$ des Petrinetz-Typs von N gilt.* *

Wir bezeichnen einen Schritt mit s und die Menge aller Schritte eines Netzes N mit der Markierung M mit S.

3.5.3 Aktivierung eines Schrittes

Wir führen nun einen weiteren globalen Parameter der Schaltregel eines Petrinetz-Typs ein. Er heißt *Auswahlparameter* und bestimmt auf einer Menge von Schritten S eines Netzes N mit Markierung M eine Halbordnung. Diese Halbordnung ist eine Relation, wird mit $\prec$ bezeichnet und durch

$$\prec\colon S \times S \to \mathbb{B} \tag{3.54}$$

definiert. Wir schreiben $\prec$ auch als Infixoperator. Zur Unterscheidung der verschiedenen Operatoren für den Auswahlparameter $\prec$ verwenden wir Indizes. Der Parameter $\prec$ bestimmt beispielsweise ob ein Schritt in einem anderen „enthalten“ ist. Da ein Schritt eine Multimenge ist, können wir für diese Eigenschaft die Teilmengenbeziehung von Multimengen (siehe Gl. (2.5a)) verwenden. Mit $s_1, s_2 \in S$ sei dann

$$s_1 \prec_{\text{sub}} s_2 \quad \text{gdw.} \quad s_1 \sqsubseteq s_2 \tag{3.55}$$

Ein anderes Beispiel ist, die Summe von Prioritätslabeln der Transitionen eines Schrittes miteinander zu vergleichen. Dazu definieren wir eine Funktion ν, die die Summe der Prioritätslabel von Schritten bestimmt. Sei L_{prio} das Prioritätslabel der Transitionen ($L_{\text{prio}} : T \to \mathbb{N}$) und sei S eine Menge von Schritten. Sei $\nu : S \to \mathbb{N}$ mit $s \in S$

$$\nu(s) = \sum_{\forall (t,\delta) \in s} s(t,\delta) \cdot L_{\text{prio}}(t) \tag{3.56}$$

die Summe der Prioritätslabel der Transitionen des Schrittes und mit $s_1, s_2 \in S$ sei

$$s_1 \prec_{\text{prio}} s_2 \quad \text{gdw.} \quad \nu(s_1) < \nu(s_2) \tag{3.57}$$

der entsprechende Auswahlparameter für die Halbordnung der Schritte.

Im Allgemeinen sind zwei Schritte einer Menge S ungeordnet, so dass mit $s_1, s_2 \in S$ der triviale Auswahlparameter $\prec_{\text{def}}$ für die Halbordnung von Schritten sei

$$\prec_{\text{def}} (s_1, s_2) = \text{falsch} \tag{3.58}$$

Wir definieren nun einen aktivierten Schritt als den Schritt zu dem es bezüglich des Auswahlparameters $\prec$ keinen „größeren“ Schritt gibt.

Definition 3.5.10 (Aktivierter Schritt). *Sei S eine Menge von Schritten. Dann ist ein $\widehat{s} \in S$ ein* aktivierter Schritt*, wenn es kein $s \in S$ mit $\widehat{s} \prec s$ gibt.* *

Durch die Definition einer Halbordnungsrelation (der Auswahlparameter $\prec$ der Schaltregel) sind im Allgemeinen mehrere Schritte einer Menge von Schritten S aktiviert. Mit dem Auswahlparameter $\prec_{\text{def}}$ (3.58) sind gar alle Schritte von S aktivierte Schritte. Im Folgenden bezeichnen wir die Menge aller aktivierten Schritte mit $\widehat{S}$.

3.5.4 Schalten

Zum Schalten wählen wir nun einen Schritt $\widehat{s}$ aus der Menge der aktivierten Schritte $\widehat{S}$ aus, der geschaltet wird. Das heißt, die Markierung M des Netzes N wird in eine Markierung M_1 so überführt, dass für alle $p \in P_{F^+}$ mit P_{F^+} definiert nach (3.50) und F^+_Δ definiert nach (3.49) und $\Theta^+ = \widehat{s}$ gilt, dass der kumulierte Effekt nach Def. 3.5.7 zulässig ist. M_1 ist dann

$$M_1(p) = \begin{cases} M''(p) & \text{wenn } p \in P_{F^+} \\ M(p) & \text{sonst} \end{cases} \tag{3.59}$$

wobei $M''(p)$ der zulässige kumulierte Effekt (Def. 3.5.7) ist. Wir schreiben dafür auch

$$M \xrightarrow{\widehat{s}} M_1 \tag{3.60}$$

Die Konstruktion der Schritte (Def. 3.5.9 und 3.5.10) stellt sicher, dass nur schaltbare (aktivierte) Schritte hergestellt werden.

Für ein dynamisches Label L gilt außerdem

$$L \overset{\widehat{s}}{\rightarrowtail} L' \tag{3.61}$$

wobei $\overset{\widehat{s}}{\rightarrowtail}$ die Schaltüberführungsfunktion aus dem Universum des Labels ist. Das heißt, mit dem Schalten eines Schrittes $\widehat{s}$ verändern sich die dynamischen Label. Eine solche Überführungsfunktion in Abhängigkeit des Schrittes muss vom Petrinetz-Typ für dynamische Label definiert werden. Beispielsweise gilt für Uhren in manchen Zeit-Petrinetzen [Sta95], dass sie zurückgestellt werden, wenn sie ein Label im Umfeld einer schaltenden Transition sind. Wir können also für zeitloses Schalten mit $T_{\widehat{s}}$ nach Gl. (3.47) ($T_{\widehat{s}} = \{t \mid \exists(t, \delta) \in \widehat{s}\}$) induktiv über alle $x \in \text{def}_{L_{\text{clock}}}$ definieren

$$L_{\text{clock}} \overset{\widehat{s}}{\rightarrowtail} L'_{\text{clock}} : L'_{\text{clock}}(x) = \begin{cases} 0 & \text{wenn } x \in \overset{\bullet}{t} \cup \{t\} \text{ für alle } t \in T_{\widehat{s}} \\ L_{\text{clock}}(x) & \text{sonst} \end{cases} \tag{3.62}$$

3.5.5 Nicht-Schalten

Für ein dynamisches Label ist in seinem Universum eine Nicht-Schalt-Überführungsfunktion zu definieren, die ausgeführt wird, wenn das Petrinetz nicht schaltet. Beispielsweise definieren wir für Uhren eines Netzes N $L_{\text{clock}} : Y \to \mathbb{N}$ ($Y \in \{P_N, T_N, F_N\}$) für alle $x \in \text{def}_{L_{\text{clock}}}$

$$L_{\text{clock}} \to L''_{\text{clock}} : L''_{\text{clock}}(x) = L_{\text{clock}}(x) + 1 \tag{3.63}$$

Diese Nicht-Schalt-Übergangsfunktion erhöht alle Uhren eines Netzes um 1.

3.5.6 Zusammenfassung

Eine Schaltregel ist ein Tupel $\Omega = (\xi, \zeta, \prec)$ eines Schrittparameters ξ, eines Abhängigkeitsparameters ζ und eines Auswahlparameters für Schritte $\prec$. Der Schrittparameter zu einer Schaltregel eines Petrinetz-Typs bestimmt, welche Struktur ein Schritt in Netzen des entsprechenden Petrinetz-Typs hat. Der Abhängigkeitsparameter bestimmt, ob eine beliebige derartige Struktur tatsächlich ein Schritt ist. Der Auswahlparameter schließlich bestimmt die Ordnung aller Schritte, so dass die maximalen Elemente dieser Ordnung die aktivierten Schritte des Netzes sind.

Mit dieser parametrisierten Schaltregel wird ein großer Teil von Schaltregeln erfasst. Eine Schaltregel beschreibt nur die Veränderung der Markierung eines Netzes und der dynamischen Label. Sie macht keine Aussagen über Eigenschaften von Abläufen wie Fairness und Progress [Rei98] oder einen Schaltzwang (z. B. in Zeit-Petrinetzen [Sta95]).

Die Grenzen des hier vorgestellten Ansatzes seien im Folgenden zusammengefasst. Die Berechnung des kumulierten Effektes unterstützt den Ansatz des *rewriting* und kein *safe delete* [DR98]. Damit ist die Unterscheidung von schwacher und starker Kapazität [DR98] nicht nötig.

Für eine Markierungsalgebra, deren Operation $\oplus$ nicht kommutativ ist (z. B. Sequenz), existiert auf Grund der Nichtkommutativität ein Nichtdeterminismus bei der Berechnung des kumulierten Effektes. In dieser Arbeit wird davon abstrahiert.

Marken sind atomar. Das heißt, sie sind unteilbar und unveränderlich. Mit diesem Ansatz kann ein dynamisches Label nicht an einer Marke erscheinen. Damit ist es nicht möglich, Marken mit Uhren zu verbinden. Solche Zeit-Petrinetze werden zwar klassifiziert [Sta95], es gibt aber keinen Hinweis auf die Benutzung eines Petrinetz-Typs mit Uhren an Marken.

Ein weiteres Paradigma unseres Ansatzes ist, dass alle Marken, mit denen ein Schritt schaltet, bereits vor dem Schalten vorhanden sind und nicht Zwischenergebnis des Schaltens sind. Aus diesem Grund haben wir *Zero Safe Nets* [BM00] als Petrinetz-Typ ausgeschlossen. Signal-Netzsysteme [Roc01] sind dagegen ein Petrinetz-Typ, da hier lediglich das Schalten einer Transition (und nicht das Ergebnis) Voraussetzung für das Schalten einer davon abhängigen Transition ist.

3.6 Petrinetz-Typen

In den vorangegangenen Abschnitten dieses Kapitels wurden die Begriffe des Petrinetz-Hyperwürfels eingeführt und diskutiert. Diese Begriffe werden im Folgenden mit ihren vorwiegend verwendeten Bezeichnern sowie den Abschnitten und Definitionen, in denen sie definiert werden, übersichtlich dargestellt.

0. Netz N (Def. 3.1.2)

1. Markenmenge A (Abschn. 3.2.1)
2. Markierungsstruktur $\mathfrak{A}$ (Abschn. 3.2.2)
3. Kantentypen Φ (Abschn. 3.4)
4. ein 4-Tupel von Mengen von Labeluniversen $(\mathfrak{L}_P, \mathfrak{L}_T, \mathfrak{L}_F, \mathfrak{L}_N)$, wobei jede Menge von Labeluniversen des 4-Tupels die Labeluniversen für Stellen $(\mathfrak{L}_P)$, Transitionen $(\mathfrak{L}_T)$, Kanten $(\mathfrak{L}_F)$ bzw. das Netz $(\mathfrak{L}_N)$ enthält. Jedes Labeluniversum $\mathfrak{l}$ ist ein Tupel $\left(X, v, k, \overset{\widehat{s}}{\rightarrowtail}, \rightarrowtail\right)$ (Def. 3.3.2, Abschn. 3.3) mit
   a) Wertebereich X
   b) Aktivierungsprädikat v
   c) Effektprädikat k
   d) Schalt-Überführungsfunktion $\overset{\widehat{s}}{\rightarrowtail}$
   e) Nicht-Schalt-Überführungsfunktion $\rightarrowtail$
5. Schaltregel Ω (Abschn. 3.5.6)
   a) Schrittparameter ξ (Abschn. 3.5.2)
   b) Abhängigkeitsparameter ζ (Abschn. 3.5.2)
   c) Auswahlparameter $\prec$ (Abschn. 3.5.3)

Diese Begriffe werden nun verwendet, um den Begriff „Petrinetz-Typ“ zu bilden. Ein Petrinetz-Typ ist ein Tupel mit 5 Parametern. Jedes Netz N ist von genau einem Petrinetz-Typ. Der Petrinetz-Typ legt mit den ersten beiden Parametern fest, welche Markenmenge in Netzen diesen Typs verwendet werden und mit Hilfe welcher Markierungsstruktur Marken zu Markierungen zusammengesetzt werden. Der dritte Parameter bestimmt die Kantentypen. Der vierte legt die Labeluniversen fest, das heißt, in Petrinetzen dieses Typs können nur Label mit einem Universum aus dem Petrinetz-Typ verwendet werden. Schließlich definiert der fünfte Parameter die Parameter der parametrisierten Schaltregel.

Definition 3.6.1 (Petrinetz-Typ). *Ein* Petrinetz-Typ *ist ein Tupel* $\left(A, \mathfrak{A}, \Phi, \widetilde{\mathfrak{L}}, \Omega\right)$ *von Parametern, wobei*

A – *eine Menge der zulässigen Marken,*

$\mathfrak{A}$ – *eine Markierungsstruktur,*

Φ – *eine Menge von Kantentypen,*

$\widetilde{\mathfrak{L}}$ – *ein Tupel von Mengen der Universen für Label für Stellen* ($\widetilde{\mathfrak{L}}_{(1)}$), *Transitionen* ($\widetilde{\mathfrak{L}}_{(2)}$), *Kanten* ($\widetilde{\mathfrak{L}}_{(3)}$) *bzw. das Netz* ($\widetilde{\mathfrak{L}}_{(4)}$),

Ω – *eine Schaltregel*

bezeichnet. *

Wir nennen einen so definierten Petrinetz-Typ auch *Instanz* des Petrinetz-Hyperwürfels, um diesen Begriff des Petrinetz-Typs von anderen abzugrenzen. Ein Petrinetz besteht aus einem Netz N und ist von genau einem Typ $\left(A, \mathfrak{A}, \Phi, \widetilde{\mathfrak{L}}, \Omega\right)$. Es hat wenigstens die drei Label $M_0 : P_N \to \mathfrak{A}(A)$ (Markierung), $\varphi : F_N \to \Phi$ (Kantentyp) und $i : F_N \to \mathfrak{A}(A \cup \mathcal{X})$ (Kanteninschrift), wobei $\mathcal{X}$ eine Menge freier Variablen mit $\mathcal{X} \cap A = \emptyset$ ist. Sinnvollerweise enthält die Menge Φ der zulässigen Kantentypen eines Petrinetz-Typs wenigstens die Kantentypen $\mathcal{F}_{\mathrm{IN}}$ und $\mathcal{F}_{\mathrm{OUT}}$ für gewöhnliche Petrinetz-Kanten.

Definition 3.6.2 (Petrinetz). *Sei* $\left(A, \mathfrak{A}, \Phi, \widetilde{\mathfrak{L}}, \Omega\right)$ *ein Petrinetz-Typ. Seien* N *ein Netz,* $\mathcal{L}$ *eine Menge von Labeln und* $\mathcal{X}$ *eine Menge von freien Variablen. Dann ist das Tupel* $(N, \mathcal{L}, \varphi, i, M_0)$ *ein* Petrinetz *vom Typ* $\left(A, \mathfrak{A}, \Phi, \widetilde{\mathfrak{L}}, \Omega\right)$, *wenn gilt*

- $A \cap \mathcal{X} = \emptyset$
- *für jedes* $L \in \mathcal{L}$ *gibt es ein* $i \in \{1, \ldots, 4\}$ *und ein* $\mathfrak{l} \in \widetilde{\mathfrak{L}}_{(i)}$, *so dass* L *ein Label des Universums* $\mathfrak{l}$ *für alle Elemente aus* def_L *ist*
- φ *ist eine Abbildung* $\varphi : F_N \to \Phi$
- i *ist eine Abbildung* $i : F_N \to \mathfrak{A}(A \cup \mathcal{X})$
- M_0 *ist eine Abbildung* $M_0 : P_N \to \mathfrak{A}(A)$ *

Für ein Petrinetz $(N, \mathcal{L}, \varphi, i, M_0)$ eines Petrinetz-Typs $\left(A, \mathfrak{A}, \Phi, \widetilde{\mathfrak{L}}, \Omega\right)$ schreiben wir

$$(N, \mathcal{L}, \varphi, i, M_0) : \left(A, \mathfrak{A}, \Phi, \widetilde{\mathfrak{L}}, \Omega\right) \tag{3.64}$$

Im Folgenden geben wir Beispiele für Petrinetz-Typen an, die wir mit Hilfe der Def. 3.6.1 darstellen.

3.6.1 „Klassische“ Petrinetz-Typen

In diesem Abschnitt werden klassische Petrinetz-Typen beschrieben. Zu Beginn dieses Kapitels wurden klassische Petrinetz-Typen informell charakterisiert. Entsprechend

der Ausführungen dort besteht ein klassischer Petrinetz-Typ lediglich aus den Kantentypen IN und OUT, d. h. der Kantentyp-Parameter klassischer Petrinetz-Typen ist $\Phi_{\text{class}} = \{\mathcal{F}_{\text{IN}}, \mathcal{F}_{\text{OUT}}\}$. Die Schaltregel der klassischen Petrinetz-Typen definiert Einzelschritte (ξ_{Sg}, Gl. (3.44)), die unabhängig voneinander sind (ζ_{def}, Gl. (3.51)) und für die der triviale Auswahlparameter ($\prec_{\text{def}}$, Gl. (3.58)) gilt. Die Schaltregel der klassischen Petrinetz-Typen besteht also aus den Parametern $\Omega_{\text{class}} = (\xi_{\text{Sg}}, \zeta_{\text{def}}, \prec_{\text{def}})$. Im Folgenden werden einige klassische Petrinetz-Typen näher beschrieben.

Elementare Netzsysteme

Elementare Netzsysteme haben genau eine Art von Marken – schwarze Marken. Der erste Parameter dieses Petrinetz-Typs ist also einelementig. Auf einer Stelle eines Elementaren Netzsystems kommt eine Marke maximal einmal vor. Die Markierungsstruktur – der zweite Parameter des Petrinetz-Typs ist die Klasse $\mathfrak{Mg}$. Eine Markierungsalgebra dieser Klasse bildet über der Menge der atomaren Elemente Teilmengen und definiert die entsprechenden Operationen für das Hinzufügen und das Entfernen dieser atomaren Elemente. Die Menge der Kantentypen und die Schaltregel sind die gleichen wie für alle klassischen Petrinetz-Typen. Außerdem werden in Elementaren Netzsystemen keine weiteren Label verwendet, so dass die Mengen der Labeluniversen leer sind. Die formale Darstellung dieses Petrinetz-Typs ist

$$\mathfrak{T}_{\text{EN}} = (\{\bullet\}, \mathfrak{Mg}, \{\mathcal{F}_{\text{IN}}, \mathcal{F}_{\text{OUT}}\}, (\emptyset, \emptyset, \emptyset, \emptyset), \Omega_{\text{class}}) \tag{3.65}$$

Für ein Petrinetz $(N, \mathcal{L}, \varphi, i, M_0) : \mathfrak{T}_{\text{EN}}$ gilt $\mathcal{L} = \emptyset$, da es keine Labeluniversen im Petrinetz-Typ gibt. Die Markierungsalgebra von Elementaren Netzsystemen ist $\mathfrak{Mg}(\{\bullet\}) = (\{\emptyset, \{\bullet\}\}, \langle \varepsilon, \oplus, \ominus \rangle)$ und $\varepsilon = \emptyset$. Damit und weiteren Folgerungen aus dem Petrinetz-Typ gilt $\varphi : F_N \to \{\mathcal{F}_{\text{IN}}, \mathcal{F}_{\text{OUT}}\}$, $M_0 : P_N \to \{\emptyset, \{\bullet\}\}$ und $i : F_N \to \{\emptyset, \{\bullet\}\} \cup \mathcal{X}$. Die Verwendung einer nicht leeren Menge freier Variablen $\mathcal{X}$ für ein Petrinetz diesen Typs hat wenig Sinn, da in einer beliebigen Belegung jedes Element von $\mathcal{X}$ in das einzige Element der Markenmenge ausgewertet wird. Auch die Verwendung einer Kanteninschrift ist überflüssig, da eine Kanteninschrift nur in eines der beiden Elemente $\emptyset$ oder $\{\bullet\}$ abbildet. Das neutrale Element $\emptyset$ entspricht einer Kante, die nicht existiert. So bleibt für Kanteninschriften nur eine sinnvolle Abbildung (die auch weggelassen werden kann).

Stellen-Transitions-Netze

Wie Elementare Netzsysteme haben Stellen-Transitions-Netze eine einelementige Markenmenge, die gewöhnlichen Kantentypen und keine weiteren Label. Die Markierungsstruktur dieses Petrinetz-Typs ist die Klasse $\mathfrak{Ms}$. Die Elemente ihrer Grundmenge sind Multimengen über der Markenmenge. Die formale Darstellung dieses

Petrinetz-Typs ist

$$\mathfrak{T}_{\mathrm{ST}} = \left(\{\bullet\}, \mathfrak{M}\mathfrak{s}, \{\mathcal{F}_{\mathrm{IN}}, \mathcal{F}_{\mathrm{OUT}}\}, (\emptyset, \emptyset, \emptyset, \emptyset), \Omega_{\mathrm{class}}\right) \tag{3.66}$$

Damit gilt für ein Petrinetz $(N, \mathcal{L}, \varphi, i, M_0) : \mathfrak{T}_{\mathrm{ST}}$, dass $\mathcal{L} = \emptyset$, $\varphi : F_N \to \{\mathcal{F}_{\mathrm{IN}}, \mathcal{F}_{\mathrm{OUT}}\}$, $M_0 : P_N \to \mathcal{B}(\{\bullet\})$ und $i : F_N \to \mathcal{B}(\{\bullet\}) \cup \mathcal{X}$. Mit derselben Argumentation wie oben ist es wenig sinnvoll in Kanteninschriften von Netzen diesen Petrinetz-Typs Variablen zu verwenden.

Coloured Petri Nets

High level Netze wie *Coloured Petri Nets* unterscheiden sich vor allem durch die Markenmenge von *low level* Netzen wie zuvor beschrieben. Die Markenmenge ist in *high level* Netzen beliebig und im Allgemeinen mehrelementig. *Coloured Petri Nets* haben als Markierungsstruktur dieselbe wie S/T-Netze – die Klasse $\mathfrak{M}\mathfrak{s}$. In der nicht erweiterten Form von *Coloured Petri Nets* [Jen92] gibt es lediglich die gewöhnlichen Kantentypen $\mathcal{F}_{\mathrm{IN}}$ und $\mathcal{F}_{\mathrm{OUT}}$. Ein *Coloured Petri Net* kann *transition guards* als statisches schaltrelevantes Label für Transitionen verwenden. Daneben gibt es weitere nicht schaltrelevante Label für die Sortenbezeichnung an Stellen und die Deklaration der Sorten, Variablen und Konstanten als Label des Netzes. Damit ist die formale Darstellung von *Coloured Petri Nets*

$$\mathfrak{T}_{\mathrm{CPN}} = \left(A, \mathfrak{M}\mathfrak{s}, \{\mathcal{F}_{\mathrm{IN}}, \mathcal{F}_{\mathrm{OUT}}\}, (\mathfrak{L}_P, \mathfrak{L}_T, \emptyset, \mathfrak{L}_N), \Omega_{\mathrm{class}}\right) \tag{3.67}$$

wobei A eine beliebige Menge von Marken ist. Die Mengen $\mathfrak{L}_P$ und $\mathfrak{L}_N$ enthalten triviale Labeluniversen der Art $\left(X, v_{\mathrm{def}}, k_{\mathrm{def}}, \overset{\widehat{s}}{\longmapsto}_{\mathrm{def}}, \longmapsto_{\mathrm{def}}\right)$ für die nicht schaltrelevanten Label der Netze dieses Typs. Die Menge $\mathfrak{L}_T$ enthält das Labeluniversum für *transition guards*

$$\mathfrak{l}_{\mathrm{guard}} = \left(O \times \mathfrak{M}\mathfrak{s}(A \cup \mathcal{X}) \times \cdots \times \mathfrak{M}\mathfrak{s}(A \cup \mathcal{X}), v_{\mathrm{guard}}, k_{\mathrm{def}}, \overset{\widehat{s}}{\longmapsto}_{\mathrm{def}}, \longmapsto_{\mathrm{def}}\right) \tag{3.68}$$

wobei für alle $o \in O$ gilt $o : \mathfrak{M}\mathfrak{s}(A \cup \mathcal{X}) \times \cdots \times \mathfrak{M}\mathfrak{s}(A \cup \mathcal{X}) \to \mathbb{B}$ und v_{guard} sei

$$v_{\mathrm{guard}}(x)(t, \delta) = x_{(1)}(\delta(x_{(2)}), \ldots, \delta(x_{(n)})) \tag{3.69}$$

FIFO-Netze

Mit dem Begriff FIFO-Netze werden die (*high level*) Petrinetz-Typen beschrieben, in denen die Markierungsstruktur eine Algebra ist, die Sequenzen beschreibt. Die formale Darstellung dieses Petrinetz-Typs ist

$$\mathfrak{T}_{\mathrm{FIFO}} = \left(A, \mathfrak{Sq}, \{\mathcal{F}_{\mathrm{IN}}, \mathcal{F}_{\mathrm{OUT}}\}, \widetilde{\mathfrak{L}}, \Omega_{\mathrm{class}}\right) \tag{3.70}$$

wobei A eine beliebige Markenmenge und $\widetilde{\mathfrak{L}}$ ein 4-Tupel beliebiger Labeluniversen ist.

3.6.2 Kapazität

In manchen Petrinetzen werden nicht nur knappe Ressourcen modelliert (beschränkte Anzahl von Marken auf einer Stelle) sondern auch die Aufnahmefähigkeit für bestimmte Ressourcen. Beispielsweise modelliert man so Speicherplatz und die Aufnahmefähigkeit eines Übertragungskanals. Diese Kapazität einer Stelle wird häufig durch ein Label einer Stelle modelliert. Obwohl in der Netztheorie eine einfache Konstruktion existiert [DR98], Kapazitäten durch Komplement-Stellen zu modellieren, werden aus Gründen der Übersichtlichkeit der Netzmodelle dennoch Kapazitätslabel verwendet.

Petrinetz-Typen mit Kapazität sind also solche $\mathfrak{T} = \left(A, \mathfrak{A}, \Phi, \widetilde{\mathfrak{L}}, \Omega\right)$, die in der Menge der Labeluniversen für Stellen $\widetilde{\mathfrak{L}}_{(1)}$ ein Labeluniversum $\mathfrak{l}_{\text{cap}} = \left(\mathfrak{A}(A), v, k_{\text{cap}}, \overset{\widehat{s}}{\rightarrowtail}, \rightarrowtail\right)$ enthalten, das ein Effektprädikat k_{cap} definiert, so dass

$$k_{\text{cap}}(m_1)(m_2) = \begin{cases} \text{wahr} & \text{wenn } c(m_1) \geq c(m_2) \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.71}$$

wobei c die Mächtigkeit der Markierung sei (Def. 3.2.1).

3.6.3 Zeit

Zeit-Petrinetze haben, wie schon in Abschn. 2.2.2 auf S. 21 ausgeführt, Uhren und Zeitanschriften als Label an Netzelementen. Eine Zeitanschrift wird bei der Berechnung eines aktivierten Schaltmodus gegen eine Uhr ausgewertet. Eine Uhr wird beim Schalten eines Schrittes zurückgesetzt, wenn sie mit dem Schritt in Verbindung gebracht wird. Außerdem wird eine Uhr unabhängig vom Schalten erhöht. Sei Y eine Menge von gleichartigen Netzelementen, dann ist eine Uhr im Allgemeinen ein Label $L_{\text{clock}} : Y \to \mathbb{N} \cup \{\perp\}$, wobei ein $n \in \mathbb{N}$ die Uhrenstellung anzeigt und $\perp$ die ausgeschaltete Uhr darstellt. Eine Zeitanschrift ist ein Zeitpunkt oder eine Dauer ($L_{\text{time-ann}} : Y \to \mathbb{N}$) oder bezeichnet ein Zeitintervall ($L_{\text{time-int}} : Y \to \mathbb{N} \times \mathbb{N}$).

Das Labeluniversum einer Uhr legt im Allgemeinen ihre Schaltüberführungsfunktion und ihre Nicht-Schaltüberführungsfunktion eines Netzes des entsprechenden Typs fest. Das Labeluniversum einer Zeitanschrift definiert neben seinem Wertebereich sein Aktivierungsprädikat. Dieses bezieht sich auf die Uhrenstellung also den Wert des Uhrenlabels L_{clock}. Als Beispiel für eine Instanz des Petrinetz-Hyperwürfels, die einen Petrinetz-Typ mit Zeit beschreibt, lehnen wir uns an *timed Petri nets* [Ram74] an.

Wir verwenden die einelementige Markenmenge $\{\bullet\}$ und als Markierungsstruktur die Algebra $\mathfrak{M}_\mathfrak{s}$. Die Kanten sind gewöhnliche Kanten (Kantentypen $\mathcal{F}_{\text{IN}}$ und $\mathcal{F}_{\text{OUT}}$), und die Schaltregel ist dieselbe wie für klassische Petrinetze. Eine Zeitanschrift und eine Uhr sind jeweils Label der Transition. Wir definieren also das Labeluniversum

für Zeitanschriften

$$\left(\mathbb{N}, v_{\text{time-ann}}, k_{\text{def}}, \overset{\widehat{s}}{\rightarrowtail}_{\text{def}}, \rightarrowtail_{\text{def}}\right) \tag{3·72}$$

wobei das Aktivierungsprädikat $v_{\text{time-ann}}$ wahr ist, wenn die Uhr (siehe unten) den Wert der Zeitanschrift erreicht hat

$$v_{\text{time-ann}}(x)(t,\delta) = \begin{cases} \text{wahr} & \text{wenn } L_{\text{clock}}(t) \in \mathbb{N} \text{ und } L_{\text{clock}}(t) \geq x \\ \text{falsch} & \text{sonst} \end{cases} \tag{3·73}$$

Das Labeluniversum der Uhren ist wie folgt definiert

$$\mathfrak{l}_{\text{clock}} = \left(\mathbb{N} \cup \bot, v_{\text{def}}, k_{\text{def}}, \overset{\widehat{s}}{\rightarrowtail}_{\text{def}}, \rightarrowtail_{\text{clock}}\right) \tag{3·74}$$

Für ein Label L_{clock} des Universums $\mathfrak{l}_{\text{clock}}$ eines Netzes N sei für alle $t \in T_N$

$$L_{\text{clock}}(t) \begin{cases} \geq 0 & \text{wenn es eine Belegung } \delta \text{ so gibt,} \\ & \text{dass } (t,\delta) \text{ konzessionierter Schaltmodus ist} \\ = \bot & \text{sonst} \end{cases} \tag{3·75}$$

Die Nicht-Schaltüberführungsfunktion $\rightarrowtail_{\text{clock}}$ erhöht jede Uhr um eins für jedes Label L mit dem Universum $\mathfrak{l}_{\text{clock}}$, für alle $x \in \text{def}_L$ sei

$$L \rightarrowtail_{\text{clock}} L' : L'(x) = \begin{cases} L(x)+1 & \text{wenn } L(x) \neq \bot \\ \bot & \text{sonst} \end{cases} \tag{3·76}$$

3.6.4 Priorität

Prioritätsrelationen der Transitionen werden in Petrinetzen verwendet, um die Auswahl eines zu schaltenden Schrittes zu steuern. Häufig werden Prioritäten in Verbindung mit Einzelschritten verwendet. Dazu erhalten Transitionen eine Kennzahl. Die Ordnung aller Kennzahlen definiert dann die Prioritätsrelation. Wir benötigen für Prioritäten in Petrinetz-Typen also ein Label für Transitionen und in der Schaltregel eine entsprechende Halbordnung für Schritte. Die Menge der Labeluniversen für Transitionen enthalten also ein Labeluniversum $\mathfrak{l}_{\text{prio}}$ mit

$$\mathfrak{l}_{\text{prio}} = \left(\mathbb{N}, v_{\text{def}}, k_{\text{def}}, \overset{\widehat{s}}{\rightarrowtail}_{\text{def}}, \rightarrowtail_{\text{def}}\right) \tag{3·77}$$

Für ein Netz N ist dann ein Label $L_{\text{prio}} : T_N \to (\mathfrak{l}_{\text{prio}})_{(1)}$ definiert. Für Schritte aus der Menge der Schritte S des Netzes definieren wir eine Funktion $\max : S \to \mathbb{N}$, so dass für ein $s \in S$

$$\max(s) = \max\left(\{L_{\text{prio}}(t) \mid (t,\delta) \in s\}\right) \tag{3·78}$$

den größten Wert der Menge von Prioritätslabeln liefert, die zu Transitionen der Schaltmodi des Schrittes s gehören. Der Auswahlparameter $\prec_{\text{mp}}$ der Schaltregel ist nun wie folgt definiert

$$s_1 \prec_{\text{mp}} s_2 \quad \text{gdw.} \quad \max(s_1) < \max(s_2) \tag{3.79}$$

Ein *high level* Petrinetz-Typ mit Prioritätslabeln an Transitionen und Schrittschaltregel, wobei die Schritte nach den maximalen Prioritätslabeln geordnet sind, ist wie folgt definiert. Die Menge A sei eine beliebige Menge von Marken

$$\mathfrak{T}_{\text{prio}} = (A, \mathfrak{M}_{\mathfrak{s}}, \{\mathcal{F}_{\text{IN}}, \mathcal{F}_{\text{OUT}}\}, (\emptyset, \{\mathfrak{l}_{\text{prio}}\}, \emptyset, \emptyset), (\xi_{\text{Ms}}, \zeta_{\text{def}}, \prec_{\text{mp}})) \tag{3.80}$$

Es ist nun leicht, einen anderen Petrinetz-Typ zu definieren, in dem in der Halbordnung der Schritte das Maximum der Prioritätslabel mit maximalen Schritten (siehe Gl. (3.55)) kombiniert wird. Der entsprechende Auswahlparameter $\prec_{\text{smp}}$ ist dann für zwei Schritte einer Menge von Schritten S ($s_1, s_2 \in S$) wie folgt definiert

$$s_1 \prec_{\text{smp}} s_2 \quad \text{gdw.} \quad \max(s_1) < \max(s_2) \text{ oder } s_1 \subseteq s_2 \tag{3.81}$$

Da aus $s_1 \subseteq s_2$ folgt $\max(s_1) \leq \max(s_2)$, gilt auch wenn $s_1 \prec_{\text{smp}} s_2$ dann gilt nicht $s_2 \prec_{\text{smp}} s_1$ und $\prec_{\text{smp}}$ definiert eine Halbordnung.

3.6.5 Signal-Netzsysteme

Signal-Netzsysteme sind *low level* Petrinetze, in denen mehrere (schwarze) Marken auf einer Stelle vorkommen dürfen. Es gibt drei Arten von Kanten, neben den gewöhnlichen Petrinetz-Kanten $\mathcal{F}_{\text{IN}}$ und $\mathcal{F}_{\text{OUT}}$ noch READ-Kanten (siehe Tab. 3.1) $\mathcal{F}_{\text{RD}}$. Außerdem hat eine Transition ein Label, in dem alle Transitionen enthalten sind, die ein Ereignissignal zu der Transition senden. Eine Transition kann nur schalten, wenn sie (wie eine Transition in einem S/T-Netz mit READ-Kanten) aktiviert ist und wenn sie in einem Schritt enthalten ist, der außerdem alle Transitionen enthält, von denen die Transition ein Ereignissignal erwartet. Wir definieren das Labeluniversum für Ereignissignaleingänge von Transitionen in Signal-Netzsystemen, wobei T die Menge aller Transitionen sei

$$\mathfrak{l}_{\text{sig}} = \left(\mathcal{P}(T), v_{\text{def}}, k_{\text{def}}, \overset{\widehat{s}}{\rightarrowtail}_{\text{def}}, \rightarrowtail_{\text{def}}\right) \tag{3.82}$$

Das entsprechende Label der Ereignissignaleingänge eines Netzes N sei $L_{\text{sig}} : T_N \rightarrow (\mathfrak{l}_{\text{sig}})_{(1)}$.

Mit dem Abhängigkeitsparameter ζ_{SN} und dem Auswahlparameter $\prec_{\text{sub}}$ der Schaltregel, wie sie in Gl. (3.52) bzw. (3.55) definiert wurden, ist der Petrinetz-Typ für Signal-Netzsysteme wie folgt definiert

$$\mathfrak{T}_{\text{SN}} = (\{\bullet\}, \mathfrak{M}_{\mathfrak{s}}, \{\mathcal{F}_{\text{IN}}, \mathcal{F}_{\text{OUT}}, \mathcal{F}_{\text{RD}}\}, (\emptyset, \{\mathfrak{l}_{\text{sig}}\}, \emptyset, \emptyset), (\xi_{\text{Ms}}, \zeta_{\text{SN}}, \prec_{\text{sub}})) \tag{3.83}$$

3.6.6 Objekt-Petrinetze

Objekt-Petrinetze [Val98] (siehe auch Abschn. 2.2.2, S. 19) bestehen aus Objektnetzen und einem Systemnetz. Objektnetze definieren wir als S/T-Netze mit einem Label an Transitionen. Ein solches Label stellt die Synchronisationsbeziehung seiner Transition dar.

$$\mathfrak{T}_{\mathrm{ON}} = (\{\bullet\}, \mathfrak{M}\mathfrak{s}, \{\mathcal{F}_{\mathrm{IN}}, \mathcal{F}_{\mathrm{OUT}}\}, (\emptyset, \{\mathfrak{l}_{\mathrm{sync}}\}, \emptyset, \emptyset), \Omega_{\mathrm{class}}) \tag{3.84}$$

Dabei hat das Labeluniversum $\mathfrak{l}_{\mathrm{sync}}$ als seinen Wertebereich eine Menge von Synchronisationsbezeichnern Z. In dieser Menge Z ist ein besonderer Bezeichner $\top$ enthalten, der eine damit bezeichnete Transition nicht synchronisiert.

Systemnetze sind dann von dem Petrinetz-Typ, der eine Menge $\mathcal{O}$ von Objektnetzen enthält

$$\mathfrak{T}_{\mathrm{OPN}} = (\mathcal{O} \cup \{\bullet\}, \mathfrak{M}\mathfrak{s}, \{\mathcal{F}_{\mathrm{IN}}, \mathcal{F}_{\mathrm{OUT}}\}, (\emptyset, \{\mathfrak{l}_{\mathrm{sync}}\}, \emptyset, \emptyset), (\xi_{\mathrm{St}}, \zeta_{\mathrm{sync}}, \prec_{\mathrm{def}})) \tag{3.85}$$

Der Schrittparameter ξ_{St} der Schaltregel ist definiert wie in Gl. (3.45). Der Abhängigkeitsparameter ζ_{sync} ist für eine Multimenge von Schaltmodi Θ^+ und einen Schaltmodus (t, δ) definiert wie folgt, wobei L_{sync} ein Label von Transitionen des Labeluniversums $\mathfrak{l}_{\mathrm{sync}}$ sei

$$\zeta_{\mathrm{sync}}(\Theta^+, (t, \delta)) = \begin{cases} \text{wahr} & \text{wenn } |\Theta^+| = 0 \text{ und } L_{\mathrm{sync}}(t) = \top \\ \text{wahr} & \text{wenn } |\Theta^+| = 1 \text{ und} \\ & \text{für } (t', \delta') \in \Theta^+ \text{ gilt } L_{\mathrm{sync}}(t) = L_{\mathrm{sync}}(t') \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.86}$$

Für ein Petrinetz $(N, \mathcal{L}, \varphi, i, M_0) : \mathfrak{T}_{\mathrm{OPN}}$ mit der Markenmenge $\mathcal{O} \cup \{\bullet\}$ sei für alle $(N', \mathcal{L}', \varphi', i', M_0') \in \mathcal{O}$ $P_{N'} \subseteq P_N$, $T_{N'} \subseteq T_N$, $F_{N'} \subseteq F_N$, $R_{N'} \subseteq R_N$, so dass N' ein Netz ist; $\mathcal{L}' \subseteq \mathcal{L}$, so dass für alle $L \in \mathcal{L}'$ $\mathrm{def}_L \in \{P_{N'}, T_{N'}, F_{N'}\}$ ist; $\varphi' \subseteq \varphi$, so dass $\mathrm{def}_{\varphi'} = F_{N'}$ ist; $i' \subseteq i$, so dass $\mathrm{def}_{i'} = F_{N'}$ und $M_0' \subseteq M_0$, so dass $\mathrm{def}_{M_0'} = P_{N'}$ ist.

4 Petrinetz-Beschreibungssprache

Kapitel 3 führte ein allgemeines Petrinetz-Typ-Konzept ein. Ein Petrinetz-Typ wird demzufolge beschrieben, indem die Label, ihre Beziehungen und Wertebereiche sowie die Schaltregel für Netze dieses Typs festgelegt werden. Der Schwerpunkt liegt dabei auf einer einheitlichen Sichtweise auf alle Petrinetze. Jedes Petrinetz besteht aus einem Netz, einer Menge von Labeln und einer Schaltregel. Die Menge der Label und die Schaltregel sind typspezifisch.

In diesem Kapitel geht es um die syntaktische Repräsentation von Petrinetzen. Die Werte der Label spielen eine Rolle, nicht jedoch die Schaltregel. Es wird ein allgemeines Konzept zur Beschreibung *aller* Petrinetze vorgestellt. Dieses Konzept ist auch für strukturierte und modularisierte Petrinetze geeignet. Strukturierung und Modularisierung sind dabei typunabhängig. Nach einer Einführung in Abschn. 4.1 wird in Abschn. 4.2 das Konzept dieser Petrinetz-Beschreibungssprache vorgestellt. In Abschnitt 4.3 werden Details der software-technischen Umsetzung erläutert; und in Abschn. 4.4 werden hilfreiche Software-Werkzeuge vorgestellt.

4.1 Einleitung

Ein Petrinetz besteht aus einem Netz (also einem Graphen) und einer Menge von Labeln. Ein Label weist jeweils einem Netzelement einen Wert zu. Dieses Konzept ist in dem Sinne allgemein, dass es für Petrinetze aller Typen angewendet wird. Es liegt nun nahe, diesen syntaktischen Ansatz als allgemeine Petrinetz-Beschreibungssprache zu verwenden. Und tatsächlich wird mit der grafischen Darstellung von Petrinetzen nichts anderes gemacht: der Graph eines Petrinetzes wird gezeichnet und die Netzelemente erhalten Beschriftungen (also Label).

Der Betrachter eines Petrinetzes weiß über das Netz im Allgemeinen mehr als das, was explizit dargestellt wird. Durch Übereinkunft und besondere syntaktische Elemente weiß er beispielsweise, welches Label des Netzes die Markierung einer Stelle darstellt oder zu welchem Petrinetz-Typ das Netz gehört. Dieses implizite Wissen soll im Folgenden mit einer zu entwickelnden Petrinetz-Beschreibungssprache explizit gemacht werden.

4.1.1 Anforderungen an eine Petrinetz-Beschreibungssprache

Eine Petrinetz-Beschreibungssprache muss so *allgemein* sein, dass alle Petrinetze, das heißt, Petrinetze aller Typen damit beschrieben werden können. Ihr erster Zweck ist es, ein Dateiformat für Petrinetz-Werkzeuge zu beschreiben. Das bedeutet auch, dass eine Petrinetz-Beschreibungssprache in dem Sinne *erweiterbar* ist, dass noch nicht repräsentierte Petrinetz-Typen leicht integriert werden können.

Eine Petrinetz-Beschreibungssprache muss einen Petrinetz-Graphen mit Knoten (Stellen und Transitionen) und Kanten repräsentieren können. Da die grafische Anordnung und Darstellung von Knoten sowie die Darstellung und der Verlauf von Kanten für das Verständnis (und die Pragmatik) eines Systemmodells eine große Rolle spielt, sind *grafische Elemente* wichtig für die Petrinetz-Beschreibungssprache. Ein Label mit seinem Wert und seiner grafischen Repräsentation ist immer einem Netzelement (oder dem Netz) zugeordnet. Eine Petrinetz-Beschreibungssprache hat diese Labelzuordnung zu berücksichtigen.

Zur Darstellung großer Petrinetz-Modelle, also solcher, die zahlreiche Netzelemente und Label enthalten, ist es üblich, das Modell zu strukturieren. Zur Reproduktion eines Petrinetz-Modells ist es erforderlich, auch die Struktur zu reproduzieren. Eine Petrinetz-Beschreibungssprache benötigt also ein Konzept zur *Strukturierung* von Petrinetzen.

Schließlich ist es für eine Datenbeschreibungssprache von Vorteil, wenn sie ohne Konvertierungen (direkt) durch einen Menschen *lesbar* ist. Das erhöht ihre Akzeptanz, da sie leichter verstanden wird und Fehler auch dem menschlichen Leser auffallen. Eine derartige Petrinetz-Beschreibungssprache wird die herkömmlichen Begriffe der Fachgemeinschaft verwenden.

Eine Petrinetz-Beschreibungssprache muss, vorhergehendes zusammenfassend,

- allgemein für Petrinetze aller Typen sein
- grafische Repräsentationen von Petrinetzen beachten
- Label ihrem Netzelement zuordnen
- Strukturierung von Petrinetzen unterstützen
- durch Menschen lesbar sein.

4.1.2 Eine Petrinetz-Beschreibungssprache als Dateiformat

Der vordringlichste Nutzen einer Petrinetz-Beschreibungssprache ist ihre Verwendung als Grammatik für das Dateiformat eines Petrinetz-Werkzeuges. Eine allgemeine Petrinetz-Beschreibungssprache liefert darüber hinaus eine Grammatik für ein allgemeines Petrinetz-Austauschformat. Ein solches Format kann zum Austausch von Petrinetzen zwischen verschiedenen Werkzeugen verwendet werden.

Petrinetze werden häufig mit Hilfe von Werkzeugen erzeugt, verwaltet und bearbeitet. Nun hat jedes Petrinetz-Werkzeug seine eigene dauerhafte Repräsentation für Petrinetze. Die Repräsentation eines Petrinetzes hängt jedoch weniger vom Werkzeug als vom Petrinetz-Typ ab. Es sollte also möglich sein, eine werkzeugunabhängige Repräsentation von Petrinetzen zu entwickeln. Eine werkzeugunabhängige Repräsentation von Petrinetzen aller Typen ist das Ziel der *Petri Net Markup Language* (PNML) [JKW00, PNM00], die Gegenstand dieses Kapitels ist.

Für die Akzeptanz einer Petrinetz-Beschreibungssprache als Grundlage eines Dateiformates für Petrinetze ist es erforderlich, dass sich ein derartiges Dateiformat leicht implementieren lässt. Dabei ist zu berücksichtigen, dass Petrinetz-Werkzeuge auf verschiedenen Plattformen und in verschiedenen Programmiersprachen implementiert werden.

Der Nutzen eines einheitlichen Dateiformates als Ergebnis einer allgemeinen Petrinetz-Beschreibungssprache ist offensichtlich. Ein solches Dateiformat dient als Austauschformat für Petrinetz-Werkzeuge.

4.1.3 Extensible Markup Language (XML)

Die Anforderungen an ein plattform- und programmiersprachenunabhängiges Dateiformat lassen sich mit der Dokumentenbeschreibungssprache XML sehr gut erfüllen. XML hat darüber hinaus einige Vorteile, die im Folgenden beschrieben werden. Zunächst werden die grundlegenden Konzepte von XML erläutert.

Die *Extensible Markup Language* (XML) ist eine heute weit verbreitete Beschreibungssprache für strukturierte Dokumente [BM98]. XML basiert auf der Idee des generischen Markup aus dem Verlagswesen. Sie greift dabei auf die *Standard Generalized Markup Language* (SGML) zurück. SGML ist die Sprache, mit deren Hilfe HTML definiert wurde. XML ist eine Teilsprache von SGML. Sie lässt solche Dokumente als XML-Dokumente zu, die (wie in HTML-Dokumenten) die charakteristischen Klammern „<“ und „>“ zur Kennzeichnung von Nichtterminalen und eine noch zu erläuternde charakteristische Struktur enthalten.

Ein strukturiertes Dokument ist beispielsweise ein beliebiger Text[1]. Ein Text wird durch einen Designer um Layout-Anweisungen ergänzt, die ein Setzer zu berücksichtigen hat. Der Designer seinerseits berücksichtigt die vom Autor beabsichtigte Struktur des Textes. Für den Autor ist es in diesem Sinne unerheblich, wie der Text (der Inhalt) gedruckt wird. Darum kümmern sich Designer (Layout) und Setzer (Satz). Der Autor wird jedoch seinen Text strukturieren in Sätze, Absätze, Abschnitte, Tabellen, Abbildungen etc. Der Designer wird jedes artgleiche Element eines so strukturierten Textes mit entsprechenden Formatierungsanweisungen (Abstände, Schriftart, Schriftgrad etc.) versehen. Der Setzer erhält so genaue Anweisungen zum Satz des Dokumentes,

[1] Zum Beispiel die vorliegende Arbeit.

ohne von seiner Struktur oder seinem Inhalt etwas verstehen zu müssen. Eine Analogie im computergestützten Textsatz ist das Layout- und Textsatzsystem LaTeX/TeX. Der Autor eines LaTeX-Dokumentes markiert Teile seines Dokumentes und verlangt damit, sie besonders darzustellen. Dabei hilft ihm die LaTeX-Eingabesprache. In einer Dokumentenklasse und weiteren Makropaketen wird (von Designern) definiert, wie die markierten Teile des Dokumentes dargestellt werden. Das Satzsystem TeX schließlich setzt das Dokument entsprechend der Anweisungen.

Übertragen auf Petrinetze heißt das, ein Modellentwickler (Autor) entwirft ein Petrinetz mit Netzelementen und Labeln. Welche Label er verwenden darf, ist durch den Petrinetz-Typ (Design) festgelegt. Wie das Netz dargestellt wird, hängt vom darstellenden Petrinetz-Werkzeug (Designer, Setzer) ab.

Ein strukturiertes Dokument bildet einen Baum mit einem Wurzelknoten, von dem weitere Teilbäume ausgehen. Die Knoten des Baumes werden in verschiedene Klassen eingeteilt: *Dokumentwurzel*, *Elementknoten*, *Attributknoten* und *Textknoten*. Eine Dokumentwurzel ist ein Elementknoten mit zusätzlichen Eigenschaften, die hier nicht weiter von Interesse sind. Jedes Dokument hat genau eine Dokumentwurzel. Jeder Knoten des Baumes ist Wurzelknoten eines Teilbaumes. Ein Elementknoten ist im Allgemeinen Wurzelknoten eines nicht leeren Teilbaumes. Attribut- und Textknoten sind immer Blätter des Dokumentenbaumes, das heißt, ihre Teilbäume sind immer leer. Attributknoten dienen der näheren Bestimmung eines Elementknotens. Textknoten enthalten beliebigen Text (Zeichenketten).

Wir bezeichnen Elementknoten auch als *XML-Elemente*. Ein XML-Element wird durch ein Paar von Start- und Endemarke gekennzeichnet. Eine Startmarke in einem XML-Dokument besteht aus den beiden Klammern „<“ und „>“ sowie davon eingeschlossen dem Bezeichner des XML-Elementes. Beispielsweise ist `<element>` die Startmarke eines XML-Elementes. In der Endemarke ist dem Bezeichner des Elementes das Zeichen „/“ vorangestellt. Für das XML-Element mit dem Bezeichner `element` ist `</element>` die Endemarke. Zwischen einer Start- und einer Endemarke werden die Nachfolgeknoten des Elementknotens – weitere XML-Elemente oder Textknoten – aufgeführt. Ein Nachfolgeknoten eines XML-Elementes heißt *Subelement*.

Attributknoten werden innerhalb der Startmarke ihres Elternknotens nach dessen Bezeichner geschrieben. Jedes XML-Attribut besteht aus einem Bezeichner und einem Wert, voneinander getrennt durch das Zeichen „=“. In dem Beispiel `<element key=value>` ist `key` der Name und `value` der Wert eines Attributes des XML-Elementes `element`. Ein XML-Element kann mehrere XML-Attribute enthalten, die durch Leerzeichen voneinander getrennt werden.

Ein Textknoten enthält beliebigen Text eines entsprechenden Kodierschemas, wie im folgenden Beispiel.

```
<text-element>
  Dies ist ein Text.
</text-element>
```

Hier ist das XML-Element `text-element` Elternknoten zu dem Textknoten mit dem angegebenen Beispieltext.

Ein XML-Element ohne Subelemente, das heißt, ein Elementknoten mit leerem Teilbaum, kann auch mit einer Marke gekennzeichnet sein, die zugleich Start- *und* Endemarke ist. Eine solche Marke hat das Zeichen „/“ am Ende des Bezeichners, wie hier: `<element/>` und wird als *leeres XML-Element* bezeichnet.

Die besondere Bedeutung von XML besteht darin, dass sie durch ihre Verwendung als Dokumentenbeschreibungssprache im *World Wide Web* sehr weit verbreitet ist. Damit gibt es für nahezu jede Programmiersprache und jede Plattform Werkzeuge zum Parsen und zur Syntaxüberprüfung von XML-Dateien sowie zur Generierung von Objektmodellen.

4.1.4 Austauschformat für Petrinetze

Die *Petri Net Markup Language* (PNML) ist eine Dokumentenbeschreibungssprache für Petrinetze. Sie basiert auf XML. Damit ist jedes PNML-Dokument auch ein XML-Dokument. Durch das allgemeine Typkonzept, das sich an das Labelkonzept (vgl. Abschn. 3.3) anlehnt und in diesem Kapitel noch erläutert wird, gelingt ein Ansatz, mit dem Petrinetze aller Typen beschrieben werden können.

Weiter oben (in Abschn. 4.1.2) wurde bereits festgestellt, dass jedes Petrinetz-Werkzeug zunächst ein eigenes Dateiformat für Petrinetze hat. Ein solches Format ist so angelegt, dass Petrinetze derjenigen Typen abgespeichert werden können, die auch vom Werkzeug bearbeitet werden. Zu unterscheiden sind dabei vor allem zwei Sichtweisen auf Petrinetze, die sich im Dateiformat widerspiegeln. Petrinetze werden einerseits (wie in der vorliegenden Arbeit) als Graphen mit zwei Arten von Knoten und Kanten dazwischen aufgefasst. Ein Dateiformat dieser Sichtweise speichert Stellen, Transitionen und Kanten als Elemente des Netzes. Eine Kante verweist jeweils auf eine Stelle und eine Transition (z. B. [CPN00a, PEP98, BKK95]). In einer anderen, selteneren Sichtweise besteht ein Petrinetz aus Transitionen und Stellen. Kanten werden implizit über den Vor- und Nachbereich von Transitionen angelegt. Ein Dateiformat dieser Sichtweise speichert Stellen und Transitionen sowie zu jeder Transition den Vor- und Nachbereich. Eine Transition verweist dazu auf die Stellen in ihrem Vorbereich und auf die Stellen in ihrem Nachbereich (z. B. [RS98, Scho00]).

Anwendern von Petrinetz-Werkzeugen reichen oftmals die Leistungsmerkmale nur eines einzigen Petrinetz-Werkzeuges nicht aus, da sich Werkzeugentwickler auch aufgrund der geringen Größe der Entwicklergruppe auf ihre Kernkompetenz konzentrieren. Ein Werkzeug-Anwender verwendet deshalb mehrere verschiedene Petrinetz-Werkzeuge, z. B. eines zur Modellierung, ein anderes zur Analyse und wiederum ein anderes als Laufzeitumgebung. Dies ist allerdings effizient nur möglich, wenn dasselbe Petrinetz (bzw. seine entsprechenden Dateien) in die verschiedenen Werkzeuge eingelesen werden kann. Das heißt, nur Petrinetz-Werkzeuge, deren Dateiformate in-

einander transformiert werden können, unterstützen einen Anwender in seinem Anliegen. Es gibt einige Beispiele für die Transformation von Petrinetzen verschiedener Dateiformate [Haa98, KO98, Wer00]. Meistens sind die verwendeten Dateiformate *open source*, so dass jeder Interessent Transformationswerkzeuge implementieren kann.

Tendenziell wird es immer wieder neue Petrinetz-Werkzeuge mit anderen Anforderungen an ein Dateiformat oder auch Änderungen an bestehenden Dateiformaten geben. Das heißt, die Entwicklung von Transformationswerkzeugen ist nicht abgeschlossen und muss immer wieder überprüft werden. Dieser Tendenz würde ein allgemeines Dateiformat für Petrinetze entgegenwirken, wenn es so konstruiert werden könnte, dass Petrinetze aller Typen darin repräsentiert würden.

Die Idee, ein allgemeines Dateiformat für Petrinetze zu entwickeln, ist nicht neu. Die *Abstract Petri Net Notation* (APNN) [BKK95] war ein erster Versuch, ein allgemeines Dateiformat für Petrinetze zu etablieren. APNN ist in dem Sinne allgemein, als dass es leicht für verschiedende Petrinetz-Typen oder grafische Informationen angepasst werden kann. Sie erfüllt auch die Anforderung nach Lesbarkeit für Menschen. Es gibt jedoch nur einen eng begrenzten Vorrat an Parsern und Objektmodell-Generatoren, die zudem für jeden neuen Petrinetz-Typ angepasst werden müssen.

Mit den Bemühungen um einen Standard für *high level* Petrinetze [Bil97] wird auch angeregt, ein standardisiertes Dateiformat für Petrinetze zu schaffen. Ein erster Auftakt dazu war ein Treffen zu XML/SGML-basierten Austauschformaten für Petrinetze [BBK+00] auf der Petrinetz-Konferenz 2000 in Århus. Dort wurde vereinbart, XML als Basis für ein standardisiertes Dateiformat für Petrinetze zu verwenden.

Ein Nachteil von XML-Dateien ist ihr relativ hoher Speicherplatzbedarf, da z. B. Schlüsselwörter immer im Klartext und meistens zweimal gespeichert werden. Kompressionsverfahren (z. B. `gzip`) bieten zwar eine gute Kompressionsrate für Textdateien, können aber den Zeitbedarf beim Parsen einer Datei nicht verringern – eher im Gegenteil. Der Vorteil bereits implementierter XML-Parser wiegt diesen Nachteil auf.

Eine andere Möglichkeit XML-Dateien kompakt zu speichern, ist die Konvertierung einer XML-basierten Spezifikation nach *Binary* XML [BXM99] oder in die Datenbeschreibungssprache *Abstract Syntax Notation One* (ASN.1) [ASN97]. *Binary* XML wurde entwickelt, um die Übertragungsgröße von XML-Dokumente z. B. für WAP-Anwendungen[2] zu verringern. ASN.1 ist ein Standard zur Repräsentation (Baum-)strukturierter Daten. Sie wird vor allem zur Spezifikation von Kommunikationsprotokollen, Sicherheitsprotokollen etc. verwendet. Sie ist mit dem Ziel entwickelt worden, Daten effizient, d. h. vor allem kompakt darzustellen. Dies macht es Menschen schwer, Daten in ASN.1 zu lesen. Inzwischen gibt es Werkzeuge[3], die XML nach ASN.1 übersetzen und umgekehrt. Das heißt, am Ende der Entwicklung einer XML-basierten

[2] WAP: *Wireless Application Protocol*.

[3] Zum Beispiel URL `http://www.trl.ibm.com/projects/xml/xss4j/docs/axt-readme.html`.

Beschreibungssprache für Petrinetze, ließe sich einfach eine kompaktere Darstellungsform zur Übertragung und Speicherung von PNML-Dokumenten nutzen.

4.2 Konzept von PNML

Die PNML wurde als generische Beschreibungssprache entwickelt. Das heißt, sie ist unabhängig vom Petrinetz-Typ des zu beschreibenden Netzes. Der Petrinetz-Typ ist vielmehr der Parameter der PNML, um Netze diesen Typs zu beschreiben.

Wir unterscheiden zunächst zwei Arten von Informationen, die mit der PNML beschrieben werden – netztypunabhängige und typspezifische. Netztypunabhängige Informationen sind solche Informationen über den Petrinetz-Graphen und seine Elemente, ihre grafische Repräsentation und die Strukturierung des Netzes. Typspezifische Informationen sind Informationen über konkrete Label und ihre Werte sowie ihre zulässige Kombination.

4.2.1 Netztypunabhängige Informationen

Im folgenden werden die Informationen genauer betrachtet, die zu jedem Petrinetz unabhängig von seinem Typ vorliegen. Wie weiter unten noch klar wird, ist das Konzept für Label allgemein und netztypunabhängig. Deshalb werden Label in diesem Abschnitt betrachtet.

Petrinetze und Netzelemente

Jedes Petrinetz enthält Stellen, Transitionen und Kanten als Netzelemente. Einheiten mehrerer Netzelemente wie Seiten und Module (siehe Abschn. 4.2.2) sind ebenfalls Netzelemente. Seiten (und Module) enthalten im Allgemeinen Referenzstellen und -transitionen. Auch dies sind Netzelemente. Ein Netzelement, das eine Stelle bzw. eine Transition bezeichnet, nennen wir wie zuvor auch Knoten. Ein Netzelement, das einen Knoten oder eine Kante bezeichnet, nennen wir auch Petrinetz-Element.

Identifikatoren

Alle Netzelemente und das Netz erhalten mit PNML je einen eindeutigen Identifikator (Bezeichner). Eine Kante nutzt beispielsweise den Identifikator einer Stelle und einer Transition, um ihre Knoten zu referenzieren.

Label

Jedes Netzelement und das Netz können eigene Label haben, um diesem Element weitere Bedeutungen zu geben (vgl. Abschn. 3.3). Die zulässigen Label und ihre zulässige Kombination werden vom Petrinetz-Typ festgelegt. (vgl. Abschn. 3.6). Wir

unterscheiden hier zwei Arten von Labeln: Anschriften und Attribute. Typischerweise ist eine Anschrift ein Label mit einem unbeschränkten (bzw. sehr großen) Wertebereich. Der Name eines Netzelementes, die Markierung einer Stelle oder der *transition guard* sind Beispiele für Anschriften. Ein Attribut ist ein Label mit einem endlichen (und kleinen) Wertebereich. Ein Beispiel dafür ist der Kantentyp, der das Attribut einer Kante ist und einen der Werte z. B. IN, OUT, READ oder INHIBITOR hat. Ein anderes Beispiel ist ein Attribut zur Klassifizierung von Knoten eines Netzes, das einen Netzwerkalgorithmus beschreibt [MM00]. Außerdem hat eine Anschrift eine eigene grafische Information, während ein Attribut keine eigene grafische Information hat. Ein Attribut beeinflusst jedoch häufig die grafische Erscheinung des Netzelementes zu dem es gehört, z. B. wird eine Inhibitorkante anders dargestellt als eine READ-Kante.

In den Abschnitten 3.1 und 3.4 wurden Kanten in Petrinetz-Graphen ohne eine implizite Richtung eingeführt. Diesem Konzept folgend werden hier Kanten auf ihre Stelle, ihre Transition und ihren Kantentyp bezogen. Da die meisten Petrinetz-Typen neben IN- und OUT-Kanten keine weiteren Kantentypen für ihre Netze zulassen, wird alternativ auch die traditionelle Sichtweise mit Ursprungs- und Zielknoten einer Kante unterstützt. Eine Kante mit dem Ursprung Stelle und dem Ziel Transition ist eine IN-Kante. Und eine Kante von einer Transition zu einer Stelle ist eine OUT-Kante.

Eine Kante hat also entweder drei Argumente (Stelle, Transition, Typ) oder zwei Argumente (Ursprung, Ziel). Nur im zweiten Fall macht ein weiteres (optionales) Attribut Sinn, das den Kantentyp spezifiziert, falls eine Kante nicht einen der Typen IN oder OUT hat.

Grafische Information

Jedes Netzelement und jede Anschrift ist mit Informationen verbunden, die seine bzw. ihre grafische Erscheinung bestimmen. Für einen Knoten eines Netzes ist das beispielsweise die Position in einem Koordinatensystem; für eine Kante ist das eine Liste von Zwischenpunkten und ein Kantenverlauf. Und für eine Anschrift ist das die relative Position zu ihrem Netzelement. Absolute und relative Positionen beziehen sich auf den Referenzpunkt eines Netzelementes bzw. einer Anschrift. Wir legen fest, dass der Referenzpunkt die Mitte der grafischen Repräsentation eines Knotens, die linke untere Ecke der grafischen Repräsentation einer Anschrift bzw. die Mitte des ersten Segmentes einer Kante ist. Der Referenzpunkt eines Netzes ist der Koordinatenursprung.

Alle Positionen beziehen sich auf ein Kartesisches Koordinatensystem (x, y), in dem die x-Achse von links nach rechts und die y-Achse von oben nach unten verläuft. Ein solcher Verlauf (der y-Achse) ist für grafische Softwarewerkzeuge üblich.

Weitere grafische Informationen, wie Farbe, Größe und Form von Netzelementen sind für diese Arbeit nicht von weiterem Interesse, können aber leicht in das vorbereitete Schema der grafischen Information eingepasst werden. Da auch die Werkzeuge, die keine grafische Information zu einem Petrinetz verwalten (z. B. die Werkzeuge INA

[RS98] und LoLA [Schoo]), durch PNML unterstützt werden sollen, kann eine grafische Information zu Netzelementen und Labeln weggelassen werden.

Werkzeugspezifische Information

Einige Petrinetz-Werkzeuge speichern zusätzliche werkzeugspezifische Informationen in Petrinetz-Dateien. Diese Informationen sind für andere Werkzeuge nicht relevant. Sie haben keinen Einfluss auf das Netz an sich sondern nur auf das Netz in Bezug auf das spezielle Werkzeug. Um solch eine werkzeuginterne Information abzulegen, kann das Netz, jedes Netzelement bzw. jedes Label mit einer werkzeugspezifischen Information ausgestattet werden. Die innere Struktur dieser werkzeugspezifischen Information ist dem Werkzeug überlassen. Wir sorgen lediglich dafür, dass werkzeugspezifische Information deutlich als solche markiert ist und mit dem erzeugenden Werkzeug in Verbindung gebracht werden kann. Damit können andere Werkzeuge diese Information leicht überspringen.

4.2.2 Strukturierung

Zur Veranschaulichung großer Petrinetze ist es unbedingt erforderlich, sie einem Betrachter strukturiert aufzubereiten. Mit einer strukturierten Aufbereitung ist nicht die Zusammenfassung von Marken in besonderen Datentypen (wie z. B. beim Entwicklungsschritt von *low level* zu *high level* Petrinetzen) gemeint, sondern die räumliche Trennung bzw. Zusammenfassung von Teilen des Petrinetzes. Häufig ist die Unterstützung von Strukturierungskonzepten ein Leistungsmerkmal von Petrinetz-Editoren.

Wir unterscheiden Strukturierungskonzepte von den verschiedenen Verfeinerungskonzepten für Petrinetze (siehe [Ruc96] für einen Überblick). Verfeinerungen werden im Entwicklungsprozess eines Petrinetzmodells verwendet. Dabei werden eine Stelle, eine Transition bzw. ein Teilnetz so durch ein Teilnetz ersetzt, dass das ursprüngliche Netz und das verfeinerte Netz unter einer bestimmten Betrachtungsweise äquivalent sind [Val79, SM83, Vog93]. Bleiben die zu verfeinernden Teile eines Netzes im verfeinerten Netz erhalten, dann kann ein Strukturierungskonzept zur Darstellung verwendet werden [Web97].

Hier werden zwei Strukturierungskonzepte vorgestellt. Das eine Konzept heißt Seitenkonzept, das andere Modulkonzept. Beim Seitenkonzept wird ein Petrinetz auf mehrere *Seiten* verteilt, um so z. B. ein „großes“ Systemmodell in mehreren „kleinen“ Teilsystemen übersichtlich darzustellen. Das aus Abb. 2.1 bekannte Erzeuger/Verbraucher-System könnte beispielsweise an den Transitionen senden und empfangen in drei Teilsysteme zerlegt werden. In Abb. 4.1 ist diese Zerlegung durch gestrichelte Linien angedeutet. Teilsystemgrenzen dürfen nur Knoten aber keine Kanten schneiden.

Jedes Teilsystem wird auf einer Seite dargestellt. Jeder Knoten, der auf einer (bzw. mehreren) Teilsystemgrenzen liegt, wird genau einem Teilsystem (also einer Seite)

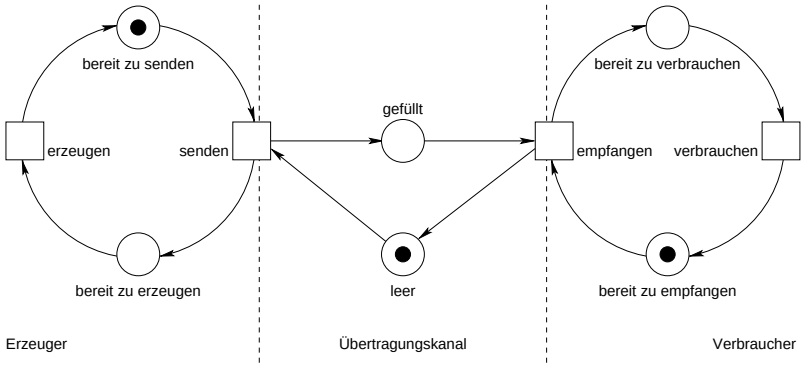


Abbildung 4.1: Zerlegung Erzeuger/Verbraucher-System in Teilsysteme

zugeordnet. In jedem weiteren Teilsystem, auf dessen Grenze der Knoten noch liegt, wird eine Kopie des Knotens dargestellt. Tatsächlich existiert der Knoten nur einmal im Petrinetz. Da Kanten nicht zerschnitten werden, gehört jede Kante zu genau einem Teilsystem, also zu genau einer Seite.

Das Seitenkonzept ist, wie hier skizziert, in einigen Petrinetz-Werkzeugen implementiert (z. B. [CPNooa, PNKoo]). Für die Semantik eines Petrinetzes ausschlaggebend ist immer das „flache“ Netz, das entsteht, wenn Seiten aufgelöst werden. Weiter unten wird dies näher erläutert.

Das zweite Konzept der Strukturierung ist das *Modulkonzept*. Ein Modul ist zunächst ein gegen seine Umgebung abgeschlossenes Petrinetz, das über eine Schnittstelle an die Umgebung angebunden werden kann. Die Umgebung hat außer auf die Schnittstelle keinen Zugriff auf Elemente des Moduls. Später werden wir sehen, wie ein Modul definiert wird und Instanzen (auch mehrere des gleichen Moduls) davon in einem Petrinetz verwendet werden. Wir geben eine Semantik zur Überführung von Modulen in Seiten an.

Ein Modul wird verwendet, wenn ein Teil eines Netzes in verschiedenen Systemmodellen (wieder-)verwendet wird, um bestimmte (Standard-)Aufgaben zu übernehmen. Eine zweite Verwendungsmöglichkeit ist die mehrfache Verwendung gleicher Teile in einem Systemmodell (vgl. [KG99]). Ein solcher Teil würde dann nur einmal als Modul implementiert aber mehrfach als Modulinstanz eingebunden werden. Eine dritte Möglichkeit ist die Verwendung von Modulen in der Systemmodellentwicklung. Zunächst wird ein Modul in einer einfachen Variante implementiert und später durch eine endgültige Variante ausgetauscht. Die gleiche Schnittstelle der einfachen und der endgültigen Variante sorgen dabei für einen konsistenten Systemenwurf (vgl. Module in Hanischs *Condition/Event-Systems* [HLoo]).

Ein Petrinetz-Werkzeug, das ein Modulkonzept implementiert, wie hier vorgestellt wird, ist bisher nicht bekannt. Module werden beispielsweise im Petrinetz-Werkzeug

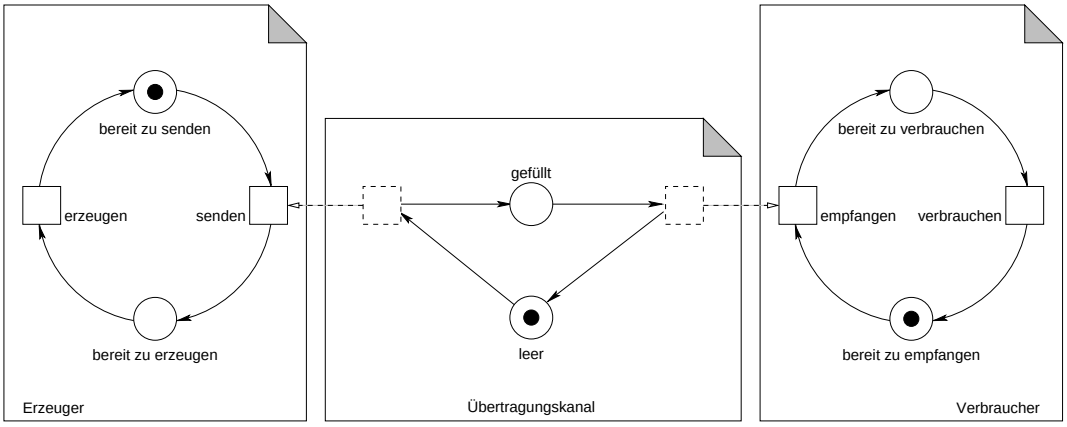


Abbildung 4.2: Petrinetz auf mehreren Seiten

Design/CPN direkt auf das Seitenkonzept zurückgeführt und als Funktionalität des Editors implementiert. Das hier betrachtete Modulkonzept ist so ähnlich für Werkzeuge der formalen Verifikation, z. B. *model checking* Werkzeuge, bekannt [McM93].

Seiten

Eine Seite eines Petrinetzes enthält eine Teilmenge der Stellen, Transitionen und Kanten des Netzes. Wir können eine Seite als eine Schnittebene des (dreidimensional dargestellten) Petrinetzes auffassen. Alle Knoten, die in diese Ebene fallen, werden auf der Seite dargestellt. Das Problem sind nun die Kanten, die nur einen ihrer Knoten auf der Seite haben. Sie können nicht ohne weiteres dargestellt werden. Deshalb führen wir für Stellen *Referenzstellen* und für Transitionen *Referenztransitionen* ein. Zusammenfassend bezeichnen wir sie auch als *Referenzknoten*. Ein Referenzknoten ist ein (grafischer) Repräsentant eines Knotens ohne eigene Petrinetz-Semantik. Das heißt, ein Referenzknoten ist kein Petrinetz-Element im Sinne der Def. 3.1.2 aber sehr wohl ein Netzelement im Sinne von Abschn. 4.2.1, da ein Referenzknoten zur Beschreibung eines strukturierten Netzes notwendig ist.

Abbildung 4.2 zeigt das Petrinetz aus Abb. 4.1 grafisch separiert auf drei Seiten entlang der skizzierten Teilsystemgrenzen. Wir unterscheiden 3 Teilsysteme (Erzeuger, Übertragungskanal, Verbraucher), die wir jeweils auf einer eigenen Seite darstellen. Die Transitionen senden und empfangen, die im Netz aus Abb. 4.1 jeweils an der Grenze zwischen zwei Teilsystemen liegen, müssen je auf einer Seite einen grafischen Repräsentanten erhalten, damit die Kanten zwischen ihnen und den Stellen des Netzes dargestellt werden können. Wir ordnen senden bzw. empfangen dem Teilsystem Erzeuger bzw. Verbraucher zu und fügen im Teilsystem Übertragungskanal je eine Referenztransition ein. Referenzknoten stellen wir grafisch durch eine gestrichelte Begrenzungslinie dar.

Jede Referenzstelle verweist auf genau eine andere Stelle, und jede Referenztransition verweist auf genau eine andere Transition. Der Verweis eines Referenzknotens auf einen anderen Knoten wird grafisch wie in Abb. 4.2 als gestrichelter Pfeil dargestellt. Wird eine Seite allein dargestellt, dann wird auf die Darstellung der Verweise zu bzw. von Knoten auf anderen Seiten verzichtet. Der Knoten, auf den verwiesen wird, kann seinerseits ein Referenzknoten sein. Eine fortlaufende Folge von Verweisen von Referenzknoten darf nicht zyklisch sein.

Die Semantik von Seiten und Referenzknoten ist recht einfach. Zunächst werden alle Seitengrenzen aufgehoben. Das heißt, alle Netzelemente außer den Seiten werden genau einer (*top level*) Seite, also dem Netz selbst, zugeordnet. Sodann werden Verweise aufgelöst. Das heißt, Referenzknoten werden durch den Knoten ersetzt, auf den ihr Verweis direkt oder indirekt zeigt. Anders ausgedrückt, löschen wir den Referenzknoten auf den kein anderer Referenzknoten verweist und verbiegen die mit ihm verbundenen Kanten zu dem Knoten, zu dem er verweist. Zu beachten ist, dass durch das Löschen von Referenzknoten auch alle Label des Referenzknoten verloren gehen. Das ist aber kein Problem, da Label an Referenzknoten keine Bedeutung haben. Im Beispiel aus Abb. 4.2 entsteht durch Auflösung der Verweise und Aufhebung der Seitengrenzen das Petrinetz aus Abb. 2.1.

Das Auflösen von Verweisen ist immer möglich, da in PNML zyklische Verweise *per definitionem* ausgeschlossen sind. Auf diese Weise ist es möglich, aus einem Netz mit Seiten und Referenzknoten ein Netz ohne Seiten- und Referenzknoten zu erzeugen. Wir verlieren lediglich grafische und strukturelle, mithin pragmatische Information aber keine semantische Information.

Seiten können neben anderen Netzelementen auch Seiten enthalten. Das ist für unsere Anschauung kein Problem, da Seitengrenzen beim Auflösen einfach ignoriert werden. Es entsteht dadurch immer ein „flaches“ Netz.

Module

In den folgenden Beispielen beschränken wir uns auf S/T-Netze. Daran soll das Prinzip der Module erklärt werden. Für andere Petrinetz-Typen gilt dasselbe Prinzip. Außerdem wird das Modulkonzept an Stellen von Petrinetzen vorgestellt. Für Transitionen gilt entsprechendes.

Abbildung 4.3 zeigt ein Beispiel für die Definition eines Moduls. Es besteht aus einer *Schnittstelle* und einer *Implementation*. Die Implementation eines Moduls enthält Netzelemente. In unserem Beispiel besteht die Implementation des Moduls aus der Referenzstelle `x` und der Stelle `y`, den Transitionen `t1` und `t2` sowie den in der Abbildung angegebenen Kanten. Die Modulgrenze, die die Schnittstelle und die Implementation umfasst, ist für Netzelemente der Implementation undurchlässig. Das heißt, Verweise auf Knoten der Implementation sind von außerhalb des Moduls nicht zulässig. Und Verweise von Referenzknoten der Implementation auf Knoten außerhalb des Moduls

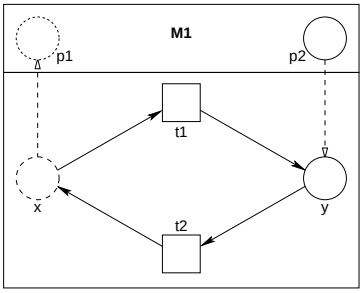


Abbildung 4.3: Das Modul M1

sind ebenso unzulässig. Für die Umgebung des Moduls ist nur die Schnittstelle des Moduls als Zugriff auf das Modul definiert. In unserem Beispiel enthält die Schnittstelle des Moduls eine Stelle p1, *Importstelle* genannt, die aus der Umgebung des Moduls importiert wird, und eine Stelle p2, *Exportstelle* genannt, die in die Umgebung exportiert wird. Ein Importknoten ist ein Parameter des Moduls, der von der Umgebung bei der Instanziierung des Moduls übergeben wird. Grafisch stellen wir einen Importknoten (ähnlich wie einen Referenzknoten) gestrichelt dar. Ein Exportknoten wird von der Instanz des Moduls der Umgebung wie ein gewöhnlicher Knoten zur Verwendung angeboten. Wir stellen einen Exportknoten grafisch wie einen gewöhnlichen Knoten dar.

Die Schnittstelle und die Implementation eines Moduls sind über Verweise, analog zu Verweisen im Seitenkonzept, miteinander verbunden. In unserem Beispiel (Abb. 4.3) verweist die Referenzstelle x auf die Importstelle p1. Genauso verweist die Exportstelle p2 auf die Stelle y. Auf einen Importknoten wird durch einen Referenzknoten der Implementation verwiesen. Verweist kein Referenzknoten der Implementation auf einen bestimmten Importknoten, dann heißt das, dass dieser Parameter in der Implementation des Moduls nicht verwendet wird. Ein Exportknoten verweist dagegen immer auf einen Knoten der Implementation.

Ein Modul wird instanziiert. Das heißt, der Schnittstelle des Moduls entsprechend wird eine Stelle an eine Importstelle und eine Transition an eine Importtransition als Parameter übergeben. Die übergebenen Knoten können auch Referenzknoten aus der Umgebung der Modulinstanz sein. Die Instanz eines Moduls stellt die Exportknoten aus der Schnittstelle des Moduls ihrer Umgebung zu Verfügung. Das heißt, auf einen solchen Knoten kann ein Referenzknoten aus der Umgebung der Modulinstanz verweisen.

Da auf die Implementation eines Moduls von außen nicht zugegriffen werden kann, genügt es, die Instanz eines Moduls durch die Schnittstelle des Moduls darzustellen. Wir nehmen an, dass die eindeutigen Identifikatoren (Bezeichner) in einem Netz keinen Punkt (.) enthalten. Abbildung 4.4 zeigt ein Netz N1, das aus drei Instanzen

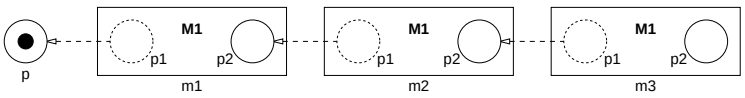


Abbildung 4.4: Netz N1 mit Instanzen des Moduls M1

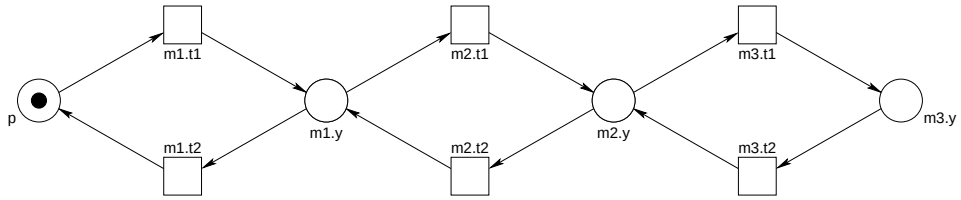


Abbildung 4.5: Semantik des Netzes N1

(m1, m2, m3) des Moduls M1 (aus Abb. 4.3) besteht. Außerdem hat das Netz eine Stelle p, markiert mit einer Marke. Die Instanz m1 übernimmt die Stelle p als aktuellen Parameter für die Importstelle p1, was wir grafisch durch einen Verweis von der Importstelle p1 der Instanz m1 zur Stelle p darstellen. Die Instanz m2 nimmt die Exportstelle p2 von m1 (geschrieben m1.p2) als ihren aktuellen Parameter p1. Und die Instanz m3 nimmt m2.p2 als aktuellen Parameter p1. Alles in allem beschreibt das modulare Netz in Abb. 4.4 das S/T-Netz in Abb. 4.5. Wir bezeichnen dieses S/T-Netz auch als die Semantik des Netzes N1.

Die Semantik von Modulen definieren wir, indem wir Module zu Seiten und Referenzknoten übersetzen. Letztendlich ersetzen wir jede Instanz eines Moduls durch eine Kopie der Implementation des Moduls.

Um das mehrfache Auftreten des gleichen Bezeichners bei der Instanziierung von Modulen zu verhindern, stellen wir den Bezeichnern eines Moduls jeweils den Bezeichner der Instanz voran. Beide Bezeichner werden durch einen Punkt voneinander getrennt. So wird aus der Stelle x des Moduls M1 nach Instanziierung beispielsweise der Instanz m1 die Stelle m1.x.

Der Ersetzungsprozess von Modulen wird rekursiv angewendet. Zunächst ersetzen wir alle Modulinstanzen innerhalb einer Modulimplementation. Dies ist nur sinnvoll, wenn es keine zyklische Abhängigkeit von Modulen gibt. Das heißt, eine Kaskade von Modulinstanzen jeweils in einer Modulimplementation muss mit einem Modul enden, das keine weiteren Module instanziiert. Außerdem lassen wir nicht zu, dass Exportknoten indirekt über Referenzknoten auf Importknoten verweisen, da wir eine kreisförmige Anordnung von Modulen wie in Abb. 4.6 nicht ausschließen. Eine kreisförmige Anordnung von Modulen *und* ein Verweis eines Exportknotens zu einem Importknoten würde einen zyklischen Verweis ergeben, den wir bei der Definition der Semantik von Seiten (siehe S. 89) ausgeschlossen hatten.

Abbildung 4.7 zeigt zwei Moduldefinitionen. Das Modul in Abb. 4.7(a) ist nicht

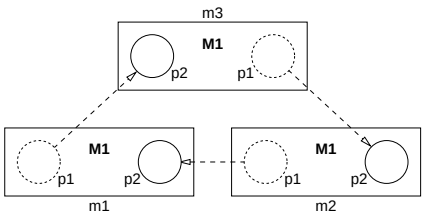
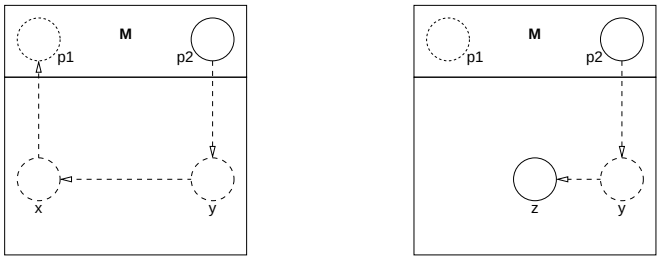


Abbildung 4.6: Ein Netz N2 mit Instanzen des Moduls M1



(a) Unzulässiges Modul

(b) Zulässiges Modul

Abbildung 4.7: Zulässige und unzulässige Moduldefinition

zulässig, da die Exportstelle p2 transitiv auf die Importstelle p1 verweist. Das Modul in Abb. 4.7(b) ist dagegen zulässig, auch wenn die Importstelle p1 in der Implementation des Moduls nicht verwendet wird. Bei der Instanziierung dieses Moduls muss jedoch für die Importstelle p1 ein aktueller Parameter übergeben werden.

Der Ersetzungsprozess von Modulen benennt zunächst alle Bezeichner konsistent um, wie oben beschrieben. Dann werden Exportknoten durch Referenzknoten ersetzt, die auf Knoten der Implementation verweisen. Importknoten werden durch Referenzknoten ersetzt, die auf Knoten außerhalb des Moduls bzw. der Seite zeigen. Knoten der Implementation, die auf Importknoten verweisen, sind bereits Referenzknoten. Sie verweisen also tatsächlich auf Knoten außerhalb des Moduls.

Das Beispiel in Abb. 4.8 zeigt das Ergebnis des Ersetzungsprozesses am Netz N1 aus Abb. 4.4. Es entstehen aus den drei Modulinstanzen drei Seiten mit jeweils drei Referenzstellen, einer Stelle y und den Transitionen t1 und t2. Durch den Prozess der Auflösung von Seiten, wie oben beschrieben, ergibt sich dann das Netz in Abb. 4.5.

Zu beachten ist, dass der Petrinetz-Typ eines Moduls in relevanten Teilen derselbe Typ ist, wie der des Netzes, das Instanzen dieses Moduls enthält. Das heißt, die im Modul verwendeten Label sind auch Label des Netzes, in das das Modul als Instanz eingeht. Sie haben die gleiche Bedeutung.

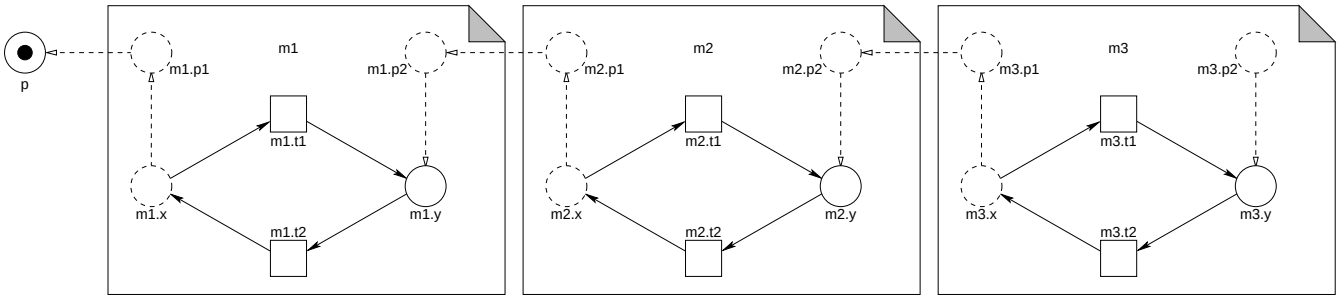


Abbildung 4.8: Modulinstanzen in Seiten überführt

Symbole

Das Modulkonzept, wie es soeben dargestellt wurde, betrifft zunächst die Netzknoten. Häufig genügt es jedoch nicht, lediglich Knoten in Modulinstanzen zu importieren oder von Modulinstanzen in die Umgebung zu exportieren. Vielmehr möchte man vor allem in *high level* Netzen auch andere Informationen in eine Modulinstanz bzw. von ihr übertragen.

Das Modul in Abb. 4.9 ist ein Beispiel für ein Modul, das weitere Informationen aus der Umgebung einer Modulinstanz aufnimmt. Es ist ein Modul für ein *high level* Netz mit Labeln für Sorten von Stellen und Variablen einer Sorte in Kanteninschriften. Sorten und Variablen werden in einem Label für eine Deklaration eines Netzes vereinbart (vgl. *Coloured Petri Nets* auf S. 15). Sie können aber auch in entsprechenden Deklarationen eines Moduls bzw. einer Seite vereinbart werden. Das Modul besteht aus der Schnittstelle mit der Importstelle `p1` und der Exportstelle `p2` sowie der Implementation aus der Referenzstelle `a` (verweist auf die Importstelle `p1`), der Transition `t`, der Stelle `b` (auf die die Exportstelle `p2` verweist) und den entsprechenden Kanten und Labeln. Außerdem wird in der Schnittstelle eine Information (`data`) übergeben, die in der Implementation verwendet wird. Wir nennen eine solche Information *Symbol* (im Falle des Moduls in Abb. 4.9 genauer *Importsymbol*) und ihre Verwendung *Referenzsymbol*.

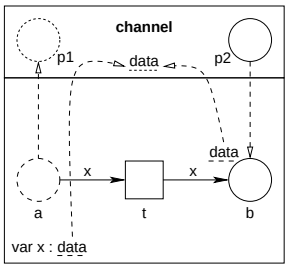


Abbildung 4.9: Modul mit Importsymbol

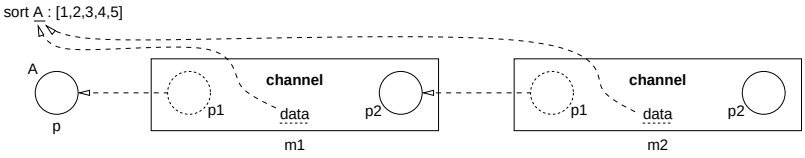


Abbildung 4.10: Instanziierung eines Moduls mit Importsymbolen (Netz N2)

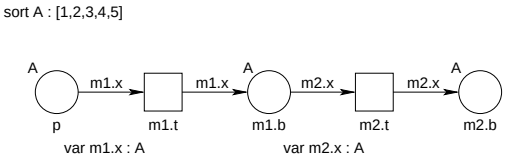


Abbildung 4.11: Semantik des Netzes N2

Ein Symbol stellen wir grafisch durchgehend unterstrichen dar. Ein Importsymbol bzw. ein Referenzsymbol wird auf verschieden unterbrochene Art unterstrichen.

Das Modul in Abb. 4.9 definiert einen einfachen Übertragungskanal für Daten einer beliebigen Sorte. Die Sorte der Daten, für die das Modul verwendet werden soll, wird bei der Instanziierung des Moduls übergeben. Dafür steht das Importsymbol data. Auf dieses Importsymbol wird in der Implementation mit Referenzsymbolen verwiesen, um die Variable x zu deklarieren und die Sorte der Stelle b festzulegen. Die Referenzstelle a und die Importstelle p1 benötigen kein Sortenlabel, da ein Referenzknoten alle Label des Knotens übernimmt, auf den er verweist.

Abbildung 4.10 zeigt die zweimalige Instanziierung des Moduls aus Abb. 4.9 in einem Netz. Im Netz wird die Sorte A deklariert – in diesem Falle als Menge der natürlichen Zahlen von 1 bis 5. Diese Sorte ist auch im Sortenlabel der Stelle p enthalten. Außerdem wird die Sorte A als Symbol des Netzes vereinbart, was grafisch durch eine Unterstreichung dargestellt wird. Den Modulinstanzen m1 und m2 des Moduls channel wird dieses Symbol als Parameter übergeben. Die Bindung der Importsymbole wird ähnlich wie für Importknoten durch einen gestrichelten Pfeil angedeutet.

Es ist unerheblich, wo in einem Netz ein Symbol vereinbart wird. In unserem Beispiel aus Abb. 4.10 könnten wir mit gutem Grund auch das Sortenlabel der Stelle p als Symbol deklarieren und als Parameter eines Moduls übergeben, da die Stelle p ohnehin Parameter einer Modulinstanz ist.

Der Ersetzungsprozess für das Netz in Abb. 4.10 führt zu dem Netz in Abb. 4.11 nachdem wir alle Bezeichner konsistent umbenannt haben, Module in Seiten transformiert haben sowie schließlich Referenzen und damit Seitengrenzen aufgelöst haben. Symbole im Sinne dieses Abschnittes werden ausschließlich bei der Instanziierung von Modulen benötigt. Die Verwendung außerhalb des Modulkonzeptes ist nicht falsch aber wenig sinnvoll, da ohnehin für die Konsistenz eines Petrinetzes und seiner Label

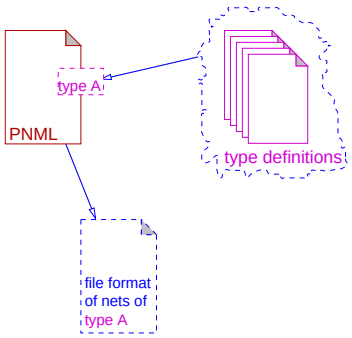


Abbildung 4.12: Zusammenspiel PNML und PNTD

gesorgt werden muss. Dafür gibt es bereits andere Methoden wie die Verwendung ausschließlich deklarierter Sorten in Sortenlabeln von Stellen. Außerdem kann das allgemeine Symbolkonzept, wie es hier vorgestellt wurde, nicht die konsistente Verwendung von Symbolen sicherstellen. Diese Aufgabe muss bei der Instanziierung eines Moduls gelöst werden.

Das Symbolkonzept von PNML benötigt keine Information über den Petrinetz-Typ, für dessen Netze weitere Parameter in Modulen neben Stellen und Transitionen verwendet werden. Das Symbolkonzept ist ebenso allgemein wie das Labelkonzept. Symbole erhalten einen eindeutigen Bezeichner, so dass auf sie verwiesen werden kann.

4.2.3 Typspezifische Informationen

In Abschn. 4.2.1 wurde bereits das Konzept für Label der Petrinetz-Beschreibungssprache PNML eingeführt. Hier geht es nun um die konkrete Definition von Labeln bzw. ihrer Darstellung in PNML-Dateien. Die Art, die Anzahl der Label sowie ihre zulässige Kombination in einem Netz wird, wie schon mehrfach betont, im Petrinetz-Typ festgelegt. Für PNML ist ein Petrinetz-Typ eine Ansammlung von Labeldefinitionen. Jede Labeldefinition legt den Wertebereich des Labels und die Art des Netzelementes fest, mit dem ein Label verbunden ist. Wir nennen eine solche Ansammlung von Labeldefinitionen *Petrinetz-Typdefinition* (PNTD).

Abbildung 4.12 beschreibt das Zusammenspiel zwischen der PNML und einer PNTD. Aus einem Vorrat an verschiedenen PNTDs wird eine ausgewählt, die einen Parameter von PNML instanziiert. Auf diese Weise wird die spezielle PNML für Netze des Typs, der durch die PNTD beschrieben wird, konstruiert.

4.2.4 Konventionen

Im Prinzip kann ein Petrinetz-Typ für PNML frei definiert werden. In Praxis möchte man jedoch gleichartige Label, wie eine Uhrenstellung oder eine Zeitintervallanschrift,

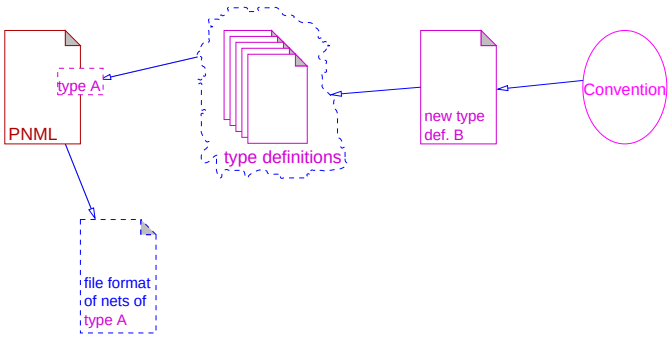


Abbildung 4.13: Vollständiges Zusammenspiel der PNML-Bestandteile

nicht für jede PNTD neu definieren. Alle oder einige Labeldefinitionen sollten deshalb zu einer Sammlung von Labeldefinitionen zusammengefasst werden. Diese Sammlung ist ein separates Dokument und wird *Konventionen* genannt. Die Konventionen garantieren, dass dieselben Label dieselbe Bedeutung für alle Petrinetz-Typen haben, in denen sie vorkommen. Dieser Ansatz erleichtert es, Netze verschiedener aber ähnlicher Typen zwischen verschiedenen Werkzeugen auszutauschen, da gleiche Label verschiedener Petrinetz-Typen dasselbe bedeuten.

Abbildung 4.13 beschreibt das Zusammenspiel aller Teile von PNML. Im linken Teil der Abbildung wird Abb. 4.12 wiederholt. Der rechte Teil beschreibt die Definition eines neuen Petrinetz-Typs, die auf den Konventionen beruht. Die benötigten Labeldefinitionen werden aus den Konventionen entnommen und in der neuen Petrinetz-Typdefinition den entsprechenden Netzelementen als Label zugeordnet. Nun kann der neue Petrinetz-Typ als Parameter für PNML verwendet werden, um Netze dieses Typs zu beschreiben.

4.3 Realisierung von PNML

Dieser Abschnitt beschreibt die software-technische Umsetzung der PNML als Dateiformat. PNML basiert auf XML und verschiedenen diesbezüglichen Hilfsmitteln. Es ist also zunächst nötig, diese Hilfsmittel einzuführen und zu erläutern.

4.3.1 XML-Schemas

In Abschn. 4.1.3 wurden die Grundelemente von XML-Dokumenten vorgestellt. XML-Dokumente eignen sich zur Beschreibung von strukturierten Daten. Ohne weitere Einschränkungen ist das, was in XML ausgedrückt werden kann, recht beliebig und wenig sinnvoll[4]. Wir benötigen ein umfassenderes Konzept, mit dem wir Klassen von

[4]Abgesehen von der strukturierten Darstellung.

Programm 4.1: EBNF einer Klasse von XML-Dokumenten

```
document ::= '<D>' ('<A/>' | ('<B/>'* '<C/>')+ ) '</D>'
```

Programm 4.2: DTD einer Klasse von XML-Dokumenten

```
<!ELEMENT D (A | (B*, C)+)>
<!ELEMENT A EMPTY>
<!ELEMENT B EMPTY>
<!ELEMENT C EMPTY>
```

XML-Dokumenten beschreiben können und an Hand dessen wir entscheiden können, ob ein beliebiges XML-Dokument dazugehört oder nicht. Ein Beispiel für eine Klasse von XML-Dokumenten ist Standard-HTML[5], die Sprache in der Web-Seiten beschrieben sind. Jedes Dokument nach Standard-HTML ist auch ein XML-Dokument.

Wir werden im Folgenden einige Beschreibungssprachen für Klassen von XML-Dokumenten näher betrachten. Zum Vergleich dieser Sprachen wird die Grammatik eines XML-Dokumentes, das in Progr. 4.1 in Erweiterter Backus-Naur-Form (EBNF) dargestellt ist, mit den Ausdrucksmitteln der entsprechenden Sprache formuliert. Die jeweiligen Schlüsselwörter der Sprache werden unterstrichen dargestellt.

Traditionell[6] werden Klassen von XML-Dokumenten mit Hilfe einer *Document Type Definition* (DTD) beschrieben. Programm 4.2 zeigt das Beispiel aus Progr. 4.1 als DTD. Vorteile von DTD sind die sehr kompakte Darstellung und ihre Anlehnung an EBNF. Nachteile von DTD sind, dass die Sprache zur Formulierung einer DTD selbst nicht XML ist und dass eine DTD nicht modular, d. h. nicht über verschiedene Dateien, aufgebaut werden kann. Der modulare Aufbau der Beschreibungssprache ist jedoch ein Grundkonzept von PNML (vgl. Abschn. 4.2).

Eine weit verbreitete Sprache zur Beschreibung von XML-Dokumentklassen ist XMLSchema [XML00]. Programm 4.3 ist die zu Progr. 4.1 äquivalente Darstellung in XMLSchema. XMLSchema wurde unter starker Beteiligung der Industrie vor allem für Internetanwendungen entwickelt. Als Datentypbeschreibungssprache, wie wir sie für PNML eher benötigen, ist XMLSchema weniger geeignet, weil sie Leistungsmerkmale enthält, die für unsere Zwecke überflüssig erscheinen. Beispielsweise wird die Häufigkeit von Elementen in einer Liste (`sequence`) mit Hilfe der XML-Attribute `minOccurs` und `maxOccurs` bei den Elementen gesteuert und ist nicht so deutlich erkennbar wie in EBNF und DTD. Außerdem war es zu Beginn der Entwicklung von PNML nicht mög-

[5] Wir unterscheiden an dieser Stelle HTML-Dokumente von den Dokumenten, die von HTML-Browsern auch interpretiert werden.

[6] Von SGML ererbt.

Programm 4.3: XMLSchema einer Klasse von XML-Dokumenten

```
<schema>
 <element name='D'>
  <complexType>
   <choice>
    <element ref='A'/>
    <sequence maxOccurs='unbounded'>
     <element ref='B' minOccurs='0' maxOccurs='unbounded'/>
     <element ref='C'/>
    </sequence>
   </choice>
  </complexType>
 </element>

 <element name='A'>
  <complexType/>
 </element>
 <element name='B'>
  <complexType/>
 </element>
 <element name='C'>
  <complexType/>
 </element>
</schema>
```

lich, die Beschreibung von PNML mit XMLSchema modular auf verschiedene Dateien aufzuteilen. Die Spezifikation von XMLSchema enthielt zwar schon zum damaligen Zeitpunkt ein Modulkonzept. Die damals zur Verfügung stehenden Werkzeuge hatten diesen Teil der Spezifikation jedoch nicht implementiert.

Der dritte Kandidat einer XML-Schema-Sprache war *Tree Regular Expressions for XML* (TREX) [Cla01]. Das Beispiel aus Progr. 4.1 als TREX-Klassenbeschreibung zeigt Progr. 4.4. Deutlich ist hier die Anlehnung von TREX an gängige Konzepte der Informatik zur Definition einer Grammatik zu sehen. Die TREX-Spezifikation war bald nach ihrer Veröffentlichung in einem Validierungswerkzeug implementiert, das XML-Dokumente gegen ihre TREX-Klassenbeschreibung überprüft. Der modulare Aufbau einer Klassenbeschreibung ist mit TREX möglich. Mit der Entwicklung von PNML durch den Autor und andere [PNMoo] wurde gezeigt, dass die Konzepte von TREX für die Spezifikation von PNML ausreichen.

In der Zwischenzeit wurde die Entwicklung von TREX mit der Entwicklung von RELAX zu RELAX NG [REL01] zusammengeführt. Die grundlegenden Konzepte, wie sie für PNML benötigt werden, blieben jedoch gleich bzw. ähnlich, so dass PNML sehr

Programm 4.4: TREX-Klassenbeschreibung von XML-Dokumenten

```
<grammar>
  <start>
    <element name="D">
      <choice>
        <ref name="A"/>
        <oneOrMore>
          <zeroOrMore>
            <ref name="B"/>
          </zeroOrMore>
          <ref name="C"/>
        </oneOrMore>
      </choice>
    </element>
  </start>

  <define name="C">
    <element name="C"><empty/></element>
  </define>
  <define name="B">
    <element name="B"><empty/></element>
  </define>
  <define name="A">
    <element name="A"><empty/></element>
  </define>
</grammar>
```

leicht auf RELAX NG übertragen werden kann[7]. Dies wird notwendig werden, wenn es neben einem Validierungswerkzeug noch weitere Softwarewerkzeuge zur Transformation bzw. zum Parsen geben wird.

Sowohl PNML als auch eine PNTD als auch die Konventionen werden mit Hilfe von TREX spezifiziert. In den folgenden Abschnitten wird die Spezifikation von PNML an Hand von Beispielen (in PNML) erläutert. Für eine PNTD wird auch der TREX-Kode gezeigt.

4.3.2 PNML

Die XML-Elemente von PNML sind nach ihren Konzepten benannt, z. B. `place` für eine Stelle, `referenceTransition` für eine Referenztransition oder `module` für eine Mo-

[7] Außerdem findet man Software zur Übersetzung einer TREX-Spezifikation in eine RELAX NG-Spezifikation

duldefinition (vgl. Abschn. 4.2). Diese XML-Elemente sind die Schlüsselwörter von PNML, und werden auch PNML-Elemente genannt. In den PNML-Beispielen werden PNML-Elemente durch Unterstreichung gekennzeichnet.

Label werden ihrer Bedeutung entsprechend bezeichnet. Sie sind, bis auf den Namen des Netzes oder eines Knotens (`name`) keine PNML-Elemente. Andersherum heißt das, dass jedes XML-Element einer PNML-Datei, das kein PNML-Element ist, ein Label sein muss und in der entsprechenden PNTD definiert sein muss. Label werden in den PNML-Beispielen nicht unterstrichen, da sie keine Schlüsselwörter von PNML sondern Schlüsselwörter einer bestimmten PNTD sind.

Das Wurzelelement einer PNML-Datei ist `pnml`. Es kann beliebig viele Elemente `net` aufnehmen. Das heißt, eine PNML-Datei kann mehrere Netze enthalten.

Netzelemente und Label

Im Folgenden betrachten wir das Netz in Abb. 4.14 und seine PNML-Darstellung in Progr. 4.5. Der Identifikator eines Petrinetzes oder eines Netzelementes wird durch das XML-Attribut `id` des PNML-Elementes angegeben. Der Wert dieses Attributes muss die Anforderungen an den Attributtyp ID von XML erfüllen (siehe [BM98]). Das heißt, der Wert muss eine Zeichenkette sein, die mit einem Buchstaben oder dem Unterstrich-Zeichen beginnt. Es können bis auf einige Ausnahmen, wie „>“, beliebige Zeichen folgen. Für das Modulkonzept von PNML fordern wir zusätzlich, dass das XML-Attribut `id` keinen Punkt (`.`) enthalten darf. Außerdem muss der Bezeichner in `id` eindeutig in Bezug auf alle Bezeichner der Datei sein. Das Netz in Abb. 4.14 enthält mehrere Netzelemente – die Stelle, die mit ready to produce bezeichnet ist, eine unbezeichnete Transition mit grob gestrichelter Darstellung, eine Kante zwischen der Stelle und der Transition sowie ein Label der Stelle, das `P` enthält, und ein Label der Kante (`x`).

Das Beispiel aus Abb. 4.14 wird in Progr. 4.5 in PNML-Kode gezeigt. Das Netz ist von einem speziellen *high level* Typ, den wir im XML-Attribut `type` des PNML-Elementes `net` entsprechend kennzeichnen (`"specHLnet"`). Das Netz hat ein Label – seinen Namen, der im Netzbild (Abb. 4.14) nicht dargestellt, aber im PNML-Kode (Progr. 4.5) enthalten ist.

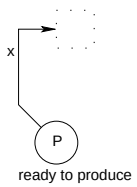


Abbildung 4.14: Ein Beispielnetz (HL-Netz)

Programm 4.5: PNML-Kode des Beispiels aus Abb. 4.14

```
<pnml>
  <net id="n1" type="specHLnet">
    <name>
      <value>Example high-level net</value>
    </name>
    <place id="p1">
      <graphics>
        <position x="20" y="40"/>
      </graphics>
      <name>
        <value>ready to produce</value>
        <graphics>
          <offset x="-10" y="10"/>
        </graphics>
      </name>
      <initialMarking>
        <value>P</value>
        <graphics>
          <offset x="-1" y="-1"/>
        </graphics>
      </initialMarking>
    </place>
    <transition id="t1">
      <graphics>
        <position x="25" y="10"/>
      </graphics>
      <toolspecific tool="PN4all" version="0.1">
        <hidden/>
      </toolspecific>
    </transition>
    <arc id="a1" source="p1" target="t1">
      <graphics>
        <position x="10" y="30"/>
        <position x="10" y="10"/>
      </graphics>
      <inscription>
        <value>x</value>
        <graphics>
          <offset x="-6" y="-16"/>
        </graphics>
      </inscription>
    </arc>
  </net>
</pnml>
```

Zum Konzept gehört es, dass jedes Label (genauer jede Anschrift) im XML-Element `value` einen Textknoten enthält, der den Wert des Labels als Zeichenkette darstellt. Im Gegensatz zu Anschriften werden Attribute von Petrinetz-Elementen als leere XML-Elemente dargestellt. Ihr XML-Attribut `value` enthält dann die Zeichenkette, die den Wert des Attributes repräsentiert (siehe Progr. 4.7).

Die Stelle des Netzes (`place`; Zeilen 6 bis 22 in Progr. 4.5) hat den dateiintern eindeutigen Identifikator *"p1"* als Wert des XML-Attributes `id`. Außerdem hat die Stelle zwei Label bzw. Anschriften. Eines enthält den Namen (`name`), das andere die initiale Markierung (`initialMarking`). Die Stelle und ihre Anschriften haben jeweils grafische Informationen, die mit `graphics` beschrieben werden. Die konkrete Definition des PNML-Elementes `graphics` hängt vom Kontext ab, in dem es auftritt. Eine Stelle bzw. ein Knoten hat eine Position (`position`) im Koordinatensystem. Während eine Anschrift einen Abstand (`offset`) zu seinem Petrinetz-Element hat.

Für die Transition (Zeilen 23 bis 30) eines Netzes gilt ähnliches wie für Stellen. Hier enthält die Transition mit dem PNML-internen Bezeichner *"t1"* neben einer grafischen Information die werkzeugspezifische Information, dass die Transition in dem Werkzeug PN4ALL der Version 0.1 als verdeckt dargestellt werden soll. Deshalb wurde die Transition in Abb. 4.14 gestrichelt gezeichnet. Werkzeugspezifische Information wird in PNML durch das PNML-Element `toolspecific` dargestellt. Dieses PNML-Element muss wenigstens die XML-Attribute `tool` und `version` besitzen. Es kann weitere XML-Elemente enthalten, die von dem Werkzeug definiert und bearbeitet werden.

Eine Kante (Zeilen 31 bis 42) verläuft von einer Quelle (XML-Attribut `source`) zu einem Ziel (`target`). Auch für eine Kante wird gefordert, dass sie einen eindeutigen Identifikator hat. Dies erleichtert Herstellern von Petrinetz-Werkzeugen die Implementation eines Parsers. Durch den Quell- und den Zielknoten sind bereits zwei grafische Punkte festgelegt. Das `graphics`-Element einer Kante enthält eine Liste weiterer Punkte, durch den der grafische Verlauf der Kante definiert wird. Die Kante unseres Beispielnetzes hat außerdem eine Anschrift (`inscription`), die die Kanteninschrift darstellt. Der grafische Abstand dieser Anschrift (PNML-Element `offset` im PNML-Element `graphics`) bezieht sich auf den Mittelpunkt des ersten Segmentes des Kantenverlaufes, das ist der Mittelpunkt zwischen der Stelle und dem ersten Zwischenpunkt der Kante.

Alternativ zu `source` und `target` kann eine Kante auch die XML-Attribute `place`, `transition` und `type` enthalten. Programm 4.6 zeigt den alternativen PNML-Kode der Kante *"a1"* aus Progr. 4.5. Diese XML-Attribute entsprechen dem Konzept der Kantentypen in Abschn. 3.4.

Der Unterschied der beiden PNML-Repräsentationen für eine Kante wird im folgenden Beispiel deutlich. Für eine übersichtliche Darstellung wird hier der PNML-Kode für die grafische Information weggelassen; ein solcher Kode könnte z. B. von einem textbasierten Werkzeug oder von einem grafischen Werkzeug, das zu Kanten keine grafischen Informationen hält und nur direkte Linienzüge zulässt, stammen. Die Pro-

Programm 4.6: Alternativer PNML-Kode einer Kante

```
<arc id="a1" place="p1" transition="t1" type="in">
  <graphics>
    <position x="10" y="30"/>
    <position x="10" y="10"/>
  </graphics>
  <inscription>
    <value>x</value>
    <graphics>
      <offset x="-6" y="-16"/>
    </graphics>
  </inscription>
</arc>
```

gramme 4.7 und 4.8 zeigen den PNML-Kode einer Inhibitorkante. Eine Inhibitorkante verläuft üblicherweise von einer Stelle (`source`) zu einer Transition (`target`). Deshalb wird eine Inhibitorkante traditionell wie in Progr. 4.7 dargestellt. Die Eigenschaft Inhibitorkante wird durch das Label (Attribut) `type` und den entsprechenden Wert seines XML-Attributes `value` dargestellt.

Die alternative Darstellung der Kante, entsprechend des Konzeptes in Abschn. 3.4, zeigt Progr. 4.8. Der Kantentyp gehört hier unbedingt als Parameter zur Kante. Dem Kantentypkonzept zufolge hat eine Kante keine Richtung sondern einen Typ (`type`), der die Aktivierungsbedingung und den Effekt bestimmt. Eine Kante verläuft zwischen einer Stelle (`place`) und einer Transition (`transition`).

Ein Label eines Netzelementes kann fehlen, wenn die PNTD des Petrinetz-Typs dies zulässt. Dann sollte klar sein, was ein fehlendes Label für das Netzelement bedeutet. Beispielsweise bedeutet eine fehlende initiale Markierung einer Stelle eines S/T-Netzes, dass die Stelle nicht markiert bzw. mit 0 markiert ist, wenn natürliche Zahlen den Wert

Programm 4.7: PNML-Kode einer Inhibitorkante

```
<arc id="a2" source="p2" target="t2">
  <type value="inhibitor"/>
</arc>
```

Programm 4.8: Alternativer PNML-Kode einer Inhibitorkante

```
<arc id="a2" place="p2" transition="t2" type="inhibitor"/>
```

der initialen Markierung angeben. Wir sprechen vom Standardwert eines Labels, wenn es eine „Übereinkunft“ gibt, was fehlende Label bedeuten. Ein anderes Beispiel sind Kantentypen wie in der Darstellung von Progr. 4.7. Eine Kante ist eine gewöhnliche Kante, wenn das Attribut `type` fehlt.

In PNML ist für ein Netz, die Stellen und die Transitionen das Label `name` vordefiniert. Es nimmt einen Bezeichner des entsprechenden Netzelementes bzw. des Netzes auf und ist nicht mit dem XML-Attribut `id` der Elemente zu verwechseln.

Seiten und Referenzknoten

Die Programme 4.9 und 4.10 zeigen zusammen den vollständigen PNML-Kode für das Netz aus Abb. 4.2. In dem Netz sind drei Seiten definiert (jeweils ab Zeile 4, 33 bzw. 58). Die Seite mit dem PNML-internen Bezeichner *"pg2"* enthält zwei Referenztransitionen (Zeilen 35 und 44). Eine Referenztransition verweist mit ihrem XML-Attribut `ref` auf eine andere Transition; in diesem Falle *"rt1"* auf *"t2"* und *"rt2"* auf *"t3"*. Die Werte der Label für die initiale Markierung einer Stelle und eine Kanteninschrift werden in diesem Petrinetz-Typ (*"PTNet"*) als natürliche Zahlen dargestellt.

Module

Programm 4.11 zeigt den PNML-Kode für das Modul `M1` aus Abb. 4.3. Neben den grafischen Informationen werden hier nun auch die Label (initiale Markierung und Kanteninschrift) weggelassen, die implizit in der Abbildung enthalten waren. Wir nehmen hier also Standardwerte für diese Label an. Das heißt, dieses Modul kann nicht instanziiert werden, wenn der Petrinetz-Typ des instanziierenden Netzes für bestimmte Label der Netzelemente keinen Standardwert festlegt.

Ein Modul besteht aus einer Schnittstelle (PNML-Element `interface`) und einer Implementation (in Progr. 4.11 alle PNML-Elemente nach `</interface>`). In dem Beispiel wird eine Importstelle (*"p1"*) und eine Exportstelle (*"p2"*) des Moduls vereinbart. Die Exportstelle verweist auf die Implementation *"y"*. Die Referenzstelle *"x"* der Implementation verweist auf die Importstelle.

Ein Modul wird in einem Netz instanziiert. Programm 4.12 zeigt den PNML-Kode des Netzes aus Abb. 4.4. Das Netz enthält Instanzen des Moduls `M1` aus Progr. 4.11. Das XML-Attribut `uri` des PNML-Elementes `instance` ist in PNML als *Uniform Resource Identifier* (URI) [XML00] definiert. Zusammen mit dem XML-Attribut `ref` verweist es auf das Modul, das instanziiert wird. Das heißt, in der mit `uri` angegebenen Datei des Internets (in diesem Falle im lokalen Pfad) befindet sich das Modul mit dem dort internen Identifikator aus dem XML-Attribut `ref`. Jeder Importknoten eines Moduls (und jedes Importsymbol) erhält einen aktuellen Parameter des Netzes. Dabei verweist das XML-Attribut `parameter` auf den internen Identifikator `id` des Importknotens des Moduls. Der übergebene Parameter ist dann ein Knoten des Netzes.

Programm 4.9: PNML-Kode für Erzeuger/Verbraucher-System

```
<pnml>
  <net id="n1" type="PTNet">
    <name><value>Erzeuger/Verbraucher-System</value></name>
    <page id="pg1">
      <name><value>Erzeuger</value></name>
      <place id="p1">
        <name><value>bereit zu erzeugen</value></name>
        <initialMarking><value>0</value></initialMarking>
      </place>
      <transition id="t1">
        <name><value>erzeugen</value></name>
      </transition>
      <place id="p2">
        <name><value>bereit zu senden</value></name>
        <initialMarking><value>1</value></initialMarking>
      </place>
      <transition id="t2">
        <name><value>senden</value></name>
      </transition>
      <arc id="a1" source="p1" target="t1">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a2" source="t1" target="p2">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a3" source="p2" target="t2">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a4" source="t2" target="p1">
        <inscription><value>1</value></inscription>
      </arc>
    </page>
    <page id="pg2">
      <name><value>Uebertragungskanal</value></name>
      <referenceTransition id="rt1" ref="t2"/>
      <place id="p5">
        <name><value>leer</value></name>
        <initialMarking><value>1</value></initialMarking>
      </place>
      <place id="p6">
        <name><value>gefuellt</value></name>
        <initialMarking><value>0</value></initialMarking>
      </place>
      <referenceTransition id="rt2" ref="t3"/>
```

Programm 4.10: PNML-Kode für Erzeuger/Verbraucher-System (Forts.)

```
      <arc id="a5" source="rt1" target="p6">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a6" source="p6" target="rt2">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a7" source="rt2" target="p5">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a8" source="p5" target="rt1">
        <inscription><value>1</value></inscription>
      </arc>
    </page>
    <page id="pg3">
      <name><value>Verbraucher</value></name>
      <place id="p3">
        <name><value>bereit zu empfangen</value></name>
        <initialMarking><value>1</value></initialMarking>
      </place>
      <transition id="t3">
        <name><value>empfangen</value></name>
      </transition>
      <place id="p4">
        <name><value>bereit zu verbrauchen</value></name>
        <initialMarking><value>0</value></initialMarking>
      </place>
      <transition id="t4">
        <name><value>verbrauchen</value></name>
      </transition>
      <arc id="a9" source="p3" target="t3">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a10" source="t3" target="p4">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a11" source="p4" target="t4">
        <inscription><value>1</value></inscription>
      </arc>
      <arc id="a12" source="t4" target="p3">
        <inscription><value>1</value></inscription>
      </arc>
    </page>
  </net>
</pnml>
```

Programm 4.11: PNML-Kode des Moduls M1

```
<module id="M1">
  <name><value>M1</value></name>
  <interface>
    <importPlace id="p1"/>
    <exportPlace id="p2" ref="y"/>
  </interface>
  <referencePlace id="x" ref="p1"/>
  <transition id="t1"/>
  <transition id="t2"/>
  <place id="y"/>
  <arc source="x" target="t1"/>
  <arc source="t1" target="y"/>
  <arc source="y" target="t2"/>
  <arc source="t2" target="x"/>
</module>
```

Programm 4.12: PNML-Kode des Netzes N1 mit Instanzen des Moduls M1

```
<net id="n1">
  <place id="p">
    <initialMarking>
      <value>1</value>
    </initialMarking>
  </place>
  <instance id="m1" ref="M1" uri="file:moduleM1.pnml">
    <importPlace parameter="p1" ref="p"/>
  </instance>
  <instance id="m2" ref="M1" uri="file:moduleM1.pnml">
    <importPlace parameter="p1" instance="m1" ref="p2"/>
  </instance>
  <instance id="m3" ref="M1" uri="file:moduleM1.pnml">
    <importPlace parameter="p1" instance="m2" ref="p2"/>
  </instance>
</net>
```

Programm 4.13: Die PNTD für S/T-Netze

```
<grammar ns="http://www.informatik.hu-berlin.de/top/pnml"
         xmlns="http://www.thaiopensource.com/trex">
  <include href="http://www.informatik.hu-berlin.de/top/pnml/pnml.trex"/>
  <include href="http://www.informatik.hu-berlin.de/top/pnml/conv.trex"/>
  <define name="NetType" combine="replace">
    <string>PTNet</string>
  </define>
  <define name="Place" combine="interleave">
    <optional>
      <ref name="InitialMarkingString"/>
    </optional>
  </define>
  <define name="Arc" combine="interleave">
    <optional>
      <ref name="InscriptionString"/>
    </optional>
  </define>
</grammar>
```

Dabei kann sich der referenzierte Knoten aus dem Bezeichner für eine Modulinstanz (`instance`) und dem Bezeichner eines Exportknotens dieser Instanz (`ref`) zusammensetzen. Fehlt das XML-Attribut `instance` des Importknotens, dann verweist `ref` auf einen Knoten des Netzes. Für Import- Exportsymbole eines Moduls gilt vergleichbares.

4.3.3 PNTD

Eine Petrinetz-Typdefinition (PNTD) definiert die spezielle PNML für Netze eines entsprechenden Typs. Dabei ordnet sie Netzelementen bzw. dem Netz Definitionen für Label zu. Wir verwenden dazu die XML-Schema-Sprache TREX. Programm 4.13 zeigt die PNTD für S/T-Netze. Für S/T-Netze werden zwei Label benötigt; eines für die initiale Markierung von Stellen und eines für Kanteninschriften. Die PNTD für S/T-Netze ist eine TREX-Datei und definiert demzufolge eine Grammatik (`grammar`). Sie beginnt mit Informationen zum Namensraum sowie dem Einbinden der Grammatiken für PNML (Zeile 3) und den Konventionen (Zeile 4, siehe Abschn. 4.3.4). Die Grammatik für PNML ist mit Hilfe von TREX so definiert, wie im vorangegangenen Abschnitt besprochen wurde, ohne an dieser Stelle im Detail auf diese Definition einzugehen. Die Konventionen enthalten Definitionen von Labeln.

In den Zeilen 5 bis 7 des Progr. 4.13 wird der Bezeichner des Petrinetz-Typs definiert, der in PNML-Dateien für S/T-Netze den Wert des XML-Attributes `type` des

PNML-Elementes <net> definiert. Dabei wird die Standarddefinition für einen Netztyp in PNML einfach ersetzt (combine="replace"). Dann wird in den Zeilen 8 bis 12 die Definition einer Stelle ("Place" aus "pnml.trex") um ein Label für initiale Markierungen erweitert (durch combine="interleave"). Der Wert des XML-Attributes combine bestimmt, dass die folgende Definition an beliebiger Stelle des zu erweiternden Elementes eingefügt werden kann. Das Label selbst wird in den Konventionen ("conv.trex") definiert[8]. Hier wird mit ref lediglich auf diese Definition verwiesen. Das TREX-Element optional deklariert das Label als ein optionales Element, das in PNML-Dateien fehlen kann. Ebenso wird die Definition einer Kante in den Zeilen 13 bis 17 um eine Inschrift erweitert.

4.3.4 Konventionen

Die Konventionen sind eine „Übereinkunft" für Labeldefinitionen. Sie können als TREX-Dokument formuliert werden. Ein solches Dokument listet Labeldefinitionen auf, ohne sie einem Netzelement zuzuordnen. Aus diesem Dokument können sie, wie im vorangegangenen Abschnitt skizziert, in Petrinetz-Typdefinitionen eingebunden werden, so dass in Netzen des entsprechenden Typs entsprechende Label verwendet werden können.

Die Definition eines speziellen Labels baut auf der Definition einer Anschrift bzw. eines Attributes auf. Deshalb zeigt das Progr. 4.14 einen Auszug aus der allgemeinen PNML-Definition. Darin werden die Elemente "Annotation" und "Attribute" definiert. Die Definition einer Anschrift ("Annotation") kann tatsächlich so verwendet werden, wenn auf eine genauere Spezifikation für den Datentyp des Wertes verzichtet wird. Dagegen dient die Definition von "Attribute" als Muster für die Definition eines Attributes. Hier wäre das Konzept eines parametrisierten Datentyps nützlich, es fehlt jedoch in TREX.

Für eine Anschrift (Zeilen 1 bis 12 von Progr. 4.14) werden zwei Subelemente definiert, die in beliebiger Reihenfolge auftreten können (TREX-Element interleave). Das Subelement <value> ist so definiert, dass es einen Textknoten (anyString) aufnimmt. Das Subelement <graphics> ist nicht zwingend (optional).

Die Definition eines Attributes besteht aus zwei Teilen (Zeilen 14-18 sowie 20-22) – dem Rahmen mit der Definition eines XML-Attributes value und dem Datentyp dieses XML-Attributes. Im Folgenden wird dies am Beispiel erläutert.

Programm 4.15 zeigt den Beginn eines Beispiels, wie die Konventionen mit Hilfe von TREX sowie der Definition für Label in PNML beschrieben werden. Es enthält (zusammen mit Progr. 4.16) alle Label, die in diesem Abschnitt für andere Beispiele verwendet wurden. Zunächst werden die Definitionen von PNML eingebunden (Zeile 3). Dann werden die Anschriften für Zeichenkettenrepräsentationen von initialen

[8] Ein Label kann auch explizit in der PNTD definiert werden. Solange es keinen allgemeinen Standard für die Konventionen gibt, ist das der vorzuziehende Weg.

Programm 4.14: Die Definition für Anschriften und Attribute in PNML

```
<define name="Annotation">
  <interleave>
    <element name="value">
      <anyString/>
    </element>
    <optional>
      <element name="graphics">
        <ref name="AnnotationGraphics"/>
      </element>
    </optional>
  </interleave>
</define>

<define name="Attribute">
  <attribute name="value">
    <ref name="AttributeValue" />
  </attribute>
</define>

<define name="AttributeValue">
  <anyString/>
</define>
```

Programm 4.15: Beispiel für ein Konventionen-Dokument

```
<grammar ns="http://www.informatik.hu-berlin.de/top/pnml"
         xmlns="http://www.thaiopensource.com/trex">
  <include href="http://www.informatik.hu-berlin.de/top/pnml/pnml.trex"/>
  <define name="InitialMarkingString">
    <element name="initialMarking">
      <ref name="Annotation"/>
    </element>
  </define>
  <define name="InscriptionString">
    <element name="inscription">
      <ref name="Annotation"/>
    </element>
  </define>
```

Programm 4.16: Beispiel für ein Konventionen-Dokument (Forts.)

```
    <define name="ArcType">
15    <element name="type">
        <attribute name="value">
          <ref name="ArcTypeValues" />
        </attribute>
      </element>
20  </define>
    <define name="ArcTypeValues">
      <choice>
        <string>normal</string>
        <string>read</string>
25      <string>inhibitor</string>
        <string>reset</string>
      </choice>
    </define>
  </grammar>
```

Markierungen und von Kanteninschriften definiert. Sie verweisen auf die Definition einer Anschrift (*"Annotation"*) in PNML. Damit repräsentieren sie Zeichenketten als die entsprechenden Label. Auf diese Definition wird in der PNTD für S/T-Netze (Progr. 4.13) verwiesen.

In Progr. 4.16 wird das Beispiel fortgesetzt. Nun werden Attribute definiert, die den Kantentyp beschreiben. Diese Definition folgt dem Muster für Attributdefinitionen aus Progr. 4.14. Es wird das PNML-Element festgelegt, das einen Kantentyp beschreibt (Zeile 15). Ein Attribut ist ein leeres XML-Element mit einem XML-Attribut `value` (Zeile 16). Als Wert eines Kantentyps (*"ArcTypeValue"*) ist dann eine der Zeichenketten der Zeilen 23 bis 26 zugelassen (TREX-Element `choice`).

An dieser Stelle kann der Autor lediglich einen Vorschlag für ein Konventionen-Dokument machen. Es ist Aufgabe einer Standardisierungskommission, insbesondere die Bezeichner und die damit dargestellten Datentypen festzulegen.

4.4 Werkzeugunterstützung

Für die XML-Schema-Sprache TREX gibt es ein Java-basiertes Werkzeug [Cla01], das XML-Dateien gegen eine TREX-Spezifikation überprüft. Damit ist es einfach möglich, eine PNML-Datei gegen eine PNTD auf syntaktische Korrektheit zu überprüfen (siehe auch [PNM00]). Das Werkzeug wird bei der Entwicklung von PNTDs und für die Syntaxkontrolle von PNML-Dateien eingesetzt.

Der Petrinetz-Kern (PNK, siehe Kap. 5) speichert Petrinetze im PNML-Format. Da

die Entwicklung von PNK und PNML jedoch parallel verläuft, unterstützt der PNK eine Nebenversion von PNML, die für die grafischen Elemente geringfügig anders definiert ist. Der PNK ist jedoch in der Lage, *jede* PNML-Datei zu lesen und in einem grafischen Editor darzustellen. Insbesondere benötigt der PNK keine Informationen über den tatsächlichen Petrinetz-Typ. Er generiert dazu einen einfachen Petrinetz-Typ, der dazu dient, Label als Zeichenketten darzustellen. Es ist offensichtlich, dass der PNK zu Netzen eines solchen Typs keine Unterstützung liefern kann, die über das Editieren, das Laden und das Speichern eines Netzes hinausgeht (näheres dazu siehe Kap. 5).

Ein wichtiges Thema für Petrinetze ist ihre grafische Darstellung. In diesem Kapitel lag der Schwerpunkt vor allem auf der Repräsentation aller Arten von Petrinetzen in PNML. Für grafische Informationen wurden nur die notwendigen Leistungsmerkmale vorgesehen. Es zeigt sich jedoch, dass PNML genau die grafischen Informationen implementiert, die auch aus dem Bereich der grafischen Visualisierung für Graphen als notwendig angesehen werden. Das heißt, PNML erreicht dieselbe Basis wie die Sprache GraphML [BEH+01], ein Austauschformat für jenen Bereich. PNML ist in diesem Sinne GraphML ähnlich.

Ein wichtiger Schwerpunkt der zukünftigen Arbeit in dem Bereich der grafischen Visualisierung ist eine Erweiterung von GraphML zum automatische Zeichnen von Graphen und zur Visualisierung. Ein erster Schritt ist dabei die Unterstützung des Standards für Vektorgrafiken SVG (*Scalable Vector Graphics*) [SVG01]. SVG ist eine Sprache zur Beschreibung zweidimensionaler Grafiken in XML. Eine Unterstützung von SVG durch GraphML kommt damit auch PNML zugute.

5 Petrinetz-Kern

Der *Petrinetz-Kern* (PNK) ist eine Infrastruktur zum Bau von Petrinetz-Werkzeugen. Er bewahrt den Programmierer eines Petrinetz-Werkzeuges davor, Standardfunktionalität für Petrinetze selbst zu implementieren. Außerdem erlaubt es der PNK seinem Nutzer, ein vorhandenes auf dem PNK basierendes Werkzeug anzupassen und zu erweitern. In diesem Kapitel werden die Ziele, die Konzepte und die Umsetzung des Petrinetz-Kerns besprochen.

Der *Nutzer des PNK* ist ein Anwendungsentwickler, der ein Petrinetz-Werkzeug herstellt. Wir unterscheiden zwei Rollen, die dieser Nutzer einnehmen kann. Zum einen ist er *PNK-Anwendungsprogrammierer*. Als solcher programmiert er mit Hilfe der Infrastruktur PNK Module von Petrinetz-Werkzeugen. Zum anderen ist ein Nutzer des PNK ein *PNK-Werkzeugbauer*. In dieser Rolle kombiniert er Module wiederum mit Hilfe der Infrastruktur PNK zu einem Petrinetz-Werkzeug. Das so entstandene Werkzeug nennen wir auch *PNK-Werkzeug*. Der *Nutzer eines PNK-Werkzeuges* spielt im Gegensatz zum Nutzer des PNK in dieser Arbeit eine untergeordnete Rolle.

In Abschn. 5.1 werden die Geschichte und Ziele des PNK sowie Hilfsmittel der Darstellung bzw. Implementierung des PNK beschrieben. Danach werden in Abschn. 5.2 die Konzepte des PNK dargestellt. Abschnitt 5.3 erläutert die Implementierung des PNK. Und schließlich stellt der Schlussabschnitt 5.4 die Implementierung des Petrinetz-Hyperwürfels (siehe Kap. 3) vor.

5.1 Einleitung

Ein Petrinetz-Werkzeug zu implementieren, ist häufig eine Zeit raubende Angelegenheit. Zudem kostet die Implementierung von *Standardfunktionalität* wie ein grafischer Editor, Visualisierung von Ergebnissen, Funktionen zum Parsen einer Datei, Abfragen, die auf den Petrinetz-Graphen bezogen sind, etc. überproportional viel Aufwand. Dieser Aufwand erscheint unnötig, da der Programmierer eines Petrinetz-Werkzeuges vor allem an der Implementierung der *Kernfunktionalität* seines Werkzeuges interessiert ist. Der Begriff der Kernfunktionalität bezeichnet hier die Funktionalität eines Werkzeuges, wegen der das Werkzeug implementiert wird. Der Aufwand zur Implementierung von Standardfunktionalität ist jedoch nötig, um erstens überhaupt ein Petrinetz-Werkzeug implementieren zu können und um zweitens beim Anwender des Petrinetz-Werkzeuges Akzeptanz zu finden.

Dieser Aufwand könnte reduziert werden, wenn *Module*, die Standardfunktionalität implementieren, in verschiedenen Werkzeugen verwendet werden könnten. Dies erfordert die Definition von Schnittstellen zwischen den Modulen und ein Konzept, wie ein Modul mit einem Petrinetz und anderen Modulen zusammenarbeitet. In dieser Sichtweise ist ein Modul, das die Kernfunktionalität eines Petrinetz-Werkzeuges implementiert, anderen Modulen in Bezug auf Schnittstellendefinition und Kooperationskonzept ähnlich. Es könnte dann in jedem anderen Petrinetz-Werkzeug wie ein Modul, das Standardfunktionalität implementiert, verwendet werden. Für diesen Ansatz wird eine *Infrastruktur* benötigt, mit deren Hilfe (Petrinetz-)Werkzeuge aus Modulen aufgebaut werden.

Der Petrinetz-Kern, der in diesem Kapitel vorgestellt wird, ist eine solche Infrastruktur zum Bau von Petrinetz-Werkzeugen [KW01, PNK00]. Er enthält verschiedene vorgefertigte Module, die den Bau des Wesentlichen eines Petrinetz-Werkzeuges unterstützen. Dazu besteht der PNK neben einer generischen Kernkomponente, die die Basisfunktionalität für Petrinetz-Werkzeuge zur Verfügung stellt, aus einem generischen Modul für einen grafischen Editor und einem generischen Modul für das allgemeine Petrinetz-Dateiformat PNML (siehe Kap. 4). Ein generisches Modul kann für alle Petrinetz-Typen ohne weitere Anpassung verwendet werden.

Dem Nutzer des PNK wird das Grundgerüst eines Petrinetz-Werkzeuges mit grafischer Benutzerschnittstelle und Dateiformat geliefert. Er muss lediglich sein spezielles Problem in einem Modul programmieren, wobei er auf eine Klassenbibliothek und eine Menge von Modulen zurückgreifen kann.

5.1.1 Entwicklungsgeschichte

Die Entwicklung des PNK begann 1996 mit ersten Ideen [KD96]. Schon bald darauf konnte die erste Version präsentiert werden [KSW97, Kin97]. Das Ziel war eine einfach zu benutzende Schnittstelle für jede Art von Funktionalität eines Petrinetz-Werkzeuges. Es sollte einfach sein, ein Petrinetz-Werkzeug durch Kombination vorgefertigter Module zu implementieren bzw. Funktionalität zu einem vorgefundenen Werkzeug hinzuzufügen.

Die ersten Versionen des PNK wurden in der objektorientierten, interpretierten Sprache Python implementiert. Diese Sprache ähnelt einer Skriptsprache und ist sehr gut zur Herstellung prototypischer Software geeignet. Erfahrene, disziplinierte Programmierer erreichen mit Python sehr schnell gute und effiziente Ergebnisse. Python hat jedoch als Programmiersprache für ein größeres Softwareprojekt einige Nachteile. So ist sie eine nicht typisierte Sprache. Das heißt, die gleiche Variable kann einmal eine Zeichenkette als Wert haben und dann eine Zahl [LF97]. Dies erfordert von den Mitgliedern eines Projektteams eine hohe Programmierdisziplin.

Für eine Infrastruktur wie den PNK ist es (wie für ein Rahmenwerk) wichtig, dass die Nutzer in derselben Sprache programmieren, in der auch die Infrastruktur ge-

schrieben wurde. Trotz ihrer Verbreitung im *open source* Bereich ist Python eine Sprache für Programmierexperten geblieben. Gelegenheitsprogrammierer, wie sich die Zielgruppe des PNK auch umschreiben lässt, zählen häufig nicht dazu. Eine Infrastruktur lebt zu einem großen Teil von der Bekanntheit und der Akzeptanz ihrer Implementationssprache.

Aus diesen Gründen wurde die anstehende Überarbeitung des Konzeptes nach der Veröffentlichung einer stabilen Python-Version des PNK [KW99, KW01] zum Anlass genommen, den PNK in Java zu reimplementieren. Java ist ebenfalls eine objektorientierte interpretierte Sprache (weitere Details in Abschn. 5.1.3), die noch besser als Python für die Zwecke des PNK geeignet ist. Im Vergleich zur Python-Version des PNK hat die Java-Version einige Verbesserungen des Konzeptes erfahren, die an dieser Stelle zusammengefasst und später in Abschn. 5.2 näher erläutert werden. Die Anwendungssteuerung des PNK implementiert nun ein flexibles Kommunikationskonzept für Anwendungsmodule des PNK. Das Labelkonzept (siehe Abschn. 3.3) wurde vereinheitlicht. Und das vorimplementierte Modul für ein Dateiformat des PNK speichert und lädt PNML-Dateien (siehe Kap. 4).

5.1.2 Unified Modeling Language (UML)

In den folgenden Abschnitten werden Konzepte des PNK mit Hilfe von Klassendiagrammen dargestellt. Zur Darstellung der Klassen und ihrer Beziehungen wird die *Unified Modeling Language* (UML) [FS98] verwendet. Vor allem die Darstellungskonzepte für Klasse, Schnittstelle, Vererbung, Assoziation und Aggregation kommen zur Anwendung.

Abbildung 5.1 zeigt ein Beispiel für die Darstellung einer Klasse Klassenname und ihrer Attribute und Methoden, in diesem Fall ein Attribut mit seinem Typ und eine Methode mit Parameterliste und Rückgabewert. Die Listen der Attribute und Methoden einer Klasse müssen nicht vollständig dargestellt werden, wenn es für den Zusammenhang nicht wesentlich ist. Eine Methode kann durch eine Abstraktionsebene (final im Beispiel) genauer spezifiziert werden. Es sind zwei Abstraktionsebenen möglich – *abstract* und *final*. Eine Methode, die als *abstract* spezifiziert wird, ist in dieser Klasse nicht implementiert und muss in konkreten abgeleiteten Klassen implementiert werden. Ein Methode, die als *final* spezifiziert wird, ist endgültig implementiert und kann in abgeleiteten Klassen nicht überschrieben werden. Jede andere (nicht näher spezi-

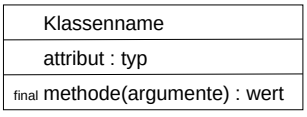


Abbildung 5.1: UML-Diagramm einer Klasse

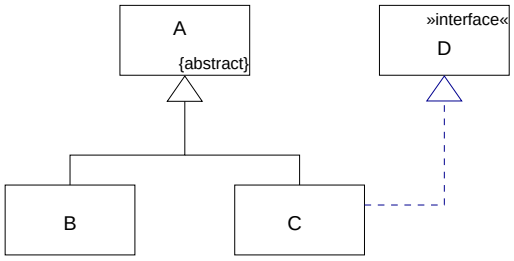


Abbildung 5.2: Vererbung (UML)

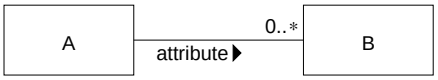


Abbildung 5.3: Assoziation (UML)

fizierte) Methode wird in der Klasse implementiert und kann in abgeleiteten Klassen überschrieben werden.

Abstrakte Klassen werden wie in Abb. 5.2 durch {abstract} und Schnittstellen durch »interface« dargestellt. Abbildung 5.2 zeigt ein Beispiel für die Vererbung von Klassen; die Klassen B und C erben von Klasse A. Außerdem zeigt die Abbildung die Darstellung für eine Implementation einer Schnittstelle; die Klasse C implementiert die Schnittstelle D.

Eine Assoziation wird zur Darstellung einer Beziehung zwischen Instanzen von (zwei) Klassen verwendet. Eine Instanz einer Klasse A steht dabei mit so vielen Instanzen einer anderen Klasse B in Beziehung, wie es das offene Intervall natürlicher Zahlen erlaubt, mit dem die Klasse B beschriftet ist ($m..n$, mit $m < n$). Ein Stern (*) steht dabei für eine beliebige obere Intervallgrenze. Sind die Intervallgrenzen gleich, wird nur eine Zahl geschrieben. Es stehen dann genau so viele Instanzen der Klasse B mit einer Instanz der Klasse A in Beziehung, wie die Zahl angibt. Im Beispiel aus Abb. 5.3 steht eine Instanz der Klasse A mit vielen (null oder mehr) Instanzen der Klasse B in Beziehung. Die Beschriftung attribute und der entsprechende Richtungspfeil zeigen dabei an, unter welchem Attribut der Klasse A diese Beziehung abgelegt ist. Ist die Beziehung dual, wenn also von einer Instanz einer Klasse A auf die entsprechende Anzahl von Instanzen einer Klasse B geschlossen werden kann *und* umgekehrt, dann erhalten beide Klassen eine entsprechende Intervallgrenze bzw. Zahl.

Eine Aggregation ist eine besondere Form der Assoziation. Sie wird für den PNK verwendet, um die Beziehung „besteht aus“ bzw. „ist enthalten in“ darzustellen. Zum Beispiel werden die Stellen eines Petrinetzes durch eine Aggregation dargestellt (ein Netz besteht aus Stellen). Im Beispiel aus Abb. 5.4 enthält eine Instanz der Klasse A wenigstens eine Instanz der Klasse B.

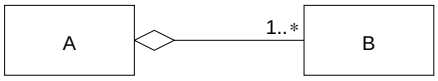


Abbildung 5.4: Aggregation (UML)

5.1.3 Java

Java ist eine objektorientierte interpretierte Sprache, deren Konzepte wohl durchdacht sind [Küh99]. Sie hat eine der Programmiersprache C ähnliche Syntax. Damit erscheint sie vielen Programmierern vertraut. Sie verzichtet auf Zeigerarithmetik, Mehrfachvererbung, Überladung von Operatoren, automatische Typkonvertierungen und ähnlich problematische Konzepte. Damit vermeidet Java Fallstricke für weniger erfahrene Programmierer. Ein Nachteil dieses Mehr an Sicherheit und Entwicklungseffizienz ist die längere Programmlaufzeit. Für die Zielgruppe des PNK ist das akzeptabel.

In die Entwicklung der Programmiersprache Java flossen Erfahrungen aus klassischen objektorientierten Sprachen wie Eiffel, SmallTalk und Objective C zusammen. Deshalb besitzt Java einen relativ kleinen Sprachumfang, der schnell erlernt werden kann, und eine umfangreiche, in der Standardvariante gut dokumentierte Klassenbibliothek. Java verwendet nur Referenzen auf Objekte. Nicht mehr referenzierte Objekte werden von der Speicherverwaltung des Java-Laufzeitsystems aus dem Speicher beseitigt.

Im Gegensatz zu Python ist Java eine streng typisierte Sprache. Die Missachtung von Typen führt zu Übersetzungs- bzw. Laufzeitfehlern, die damit den Programmierer zu einer sauberen Arbeit zwingen. Inzwischen ist Java eine sehr weit verbreitete Sprache. Es gibt Java-Interpreter für nahezu jede Plattform und zahlreiche Entwicklungsumgebungen. Für den PNK und seine Konzepte ist Java damit eine vorzügliche Basis.

5.1.4 Ziele und Prinzipien des PNK

Das Ziel des PNK, eine Infrastruktur zum Bau von Petrinetz-Werkzeugen zu sein, blieb bis hierher vage. Eine Abgrenzung zu anderen Projekten des gleichen oder eines ähnlichen Gebietes mit ähnlicher Zielstellung ist notwendig. Die Zielgruppe des PNK sind die Petrinetz-Experten, die nicht notwendigerweise Experten im Gebiet der Software-Technik sind, aber dennoch ein Petrinetz-Werkzeug herstellen (wollen). In der Software-Technik gibt es verschiedene Werkzeuge und Techniken, die es erlauben, eine Software aus Komponenten aufzubauen und diese Komponenten in verschiedenen Projekten wiederzuverwenden. Rahmenwerke (*frameworks*), komponentenbasierte Software, *meta computing* und andere sind Schlagworte aus diesem Bereich. Diese

Techniken betreffen alle Arten von Software und sind nicht auf Petrinetz-Werkzeuge beschränkt.

Für den PNK führen wir den Begriff der *Infrastruktur* in Abgrenzung zu den oben genannten Begriffen der Software-Technik ein. Mit diesem Begriff wird ausgedrückt, dass der PNK weniger als ein Rahmenwerk (*framework*) [Balo0] aber mehr als eine „einfache“ Klassenbibliothek ist. Vom Anwender eines Rahmenwerkes wird erwartet, dass er Schnittstellen implementiert und Klassen ableitet, um die Funktionalität eines Software-Werkzeuges zu implementieren. Dieser Aspekt spielt bei einer Infrastruktur zwar eine wichtige, z. B. bei Bau neuer Module, aber dennoch untergeordnete Rolle. Wichtiger ist hier, dass Module zu Werkzeugen kombiniert werden. Module werden unter der Ausnutzung der Klassenbibliothek programmiert, die eine Infrastruktur ebenfalls bereitstellt. Ein Rahmenwerk richtet sich eher an erfahrene Programmierer, die das gleiche Rahmenwerk zur Lösung verschiedener Aufgaben verwenden. Im Allgemeinen bedeutet die Verwendung eines Rahmenwerkes einen hohen Einarbeitungsaufwand. Eine Infrastruktur richtet sich dagegen an den weniger erfahrenen Programmierer, der vorzugsweise ein Werkzeug implementieren will, das schrittweise um weitere Funktionalität erweitert wird. Ein funktionsfähiges (Basis-)Werkzeug ist idealerweise ohne Programmieraufwand herzustellen.

Zur Zeit gibt es viele verschiedene Werkzeuge, die je verschiedene formale Methoden zur automatischen Verifikation von Systemen unterstützen. VeriTech [KG99] ist ein Projekt, das zum Ziel hat, Systemmodelle und Analyseresultate zwischen allen derartigen Werkzeugen auszutauschen. Ein ähnliches Ziel verfolgt das Projekt ETI [SMB97]. ETI ist eine Plattform zur Evaluierung formaler Methoden und entsprechenden Werkzeugen. Zur Zeit konzentriert sich ETI auf *model checking* Werkzeuge. Die Plattform soll Nicht-Experten auf dem Gebiet formaler Methoden helfen, die für die eigenen Bedürfnisse richtige Methode zu finden. Ein vergleichbares aber weniger ambitioniertes Projekt als die vorher genannten ist das Model-Checking Kit [MCK99]. Mehrere verschieden *model checking* Werkzeuge werden unter einer Oberfläche zusammengefasst und mit einer einheitlichen Eingabesprache bedient.

Der Petrinetz-Kern ist weniger ambitioniert als diese Projekte, weil er auf Petrinetz-Werkzeuge beschränkt ist. Außerdem verfolgt das Entwicklungsteam des PNK nicht die Intension, alle verfügbaren Petrinetz-Werkzeuge im PNK zu integrieren. Vielmehr stellt der PNK u. a. eine Plattform für jene dar, die diese Integration leisten wollen. Der PNK sorgt für eine Verbindung verschiedener Komponenten zu einem Petrinetz-Werkzeug mit einer grafischen Benutzerschnittstelle.

Das Gegenstück zu solchen Projekten wie VeriTech und ETI sind Werkzeuge, die soviel Funktionalität in einem einzigen Werkzeug integrieren wie möglich. Design/CPN [CPN00a] und PEP [PEP98] sind erfolgreiche Beispiele für solche Petrinetz-Werkzeuge. Der PNK ist kein Konkurrent zu derartigen Werkzeugen. Er wird benutzt, wenn ein bestimmter Petrinetz-Typ oder eine bestimmte Analysemethode nicht durch ein ent-

sprechendes Werkzeug unterstützt wird. Außerdem können Experimente mit dem PNK helfen, künftige Versionen monolithischer Werkzeuge zu entwickeln.

Projekte mit einer ähnlichen Zielrichtung wie der PNK sind APNN-Toolbox [APN01], CPN-AMI [CPN00b] und Maria [Mar98]. Die APNN-Toolbox ist eine offene Werkzeugsammlung, die auf dem Austauschformat APNN [BKK95] basiert. Sie konzentriert sich vor allem auf effiziente Methoden, die den Zustandsraum vor allem stochastischer Systeme analysieren. CPN-AMI ist eine Entwicklungsumgebung für Petrinetz-Werkzeuge, die die Integration verschiedener Petrinetz-Werkzeuge unter einer einheitlichen grafischen Benutzerschnittstelle unterstützt. Der zu Grunde liegende Formalismus ist nicht auf Petrinetze beschränkt und auch nicht besonders darauf ausgerichtet. Maria ist eine Klassenbibliothek von Analysealgorithmen vor allem für S/T-Netze. Der PNK ist nicht auf einige Petrinetz-Typen beschränkt, sondern für alle Petrinetz-Typen einsetzbar. Er unterstützt eine grafische Benutzerschnittstelle. Die Integration anderer Petrinetz-Werkzeuge ist möglich, spielt aber eine untergeordnete Rolle. Stattdessen kombiniert der PNK-Nutzer eigene Algorithmen mit Algorithmen aus einer Klassenbibliothek.

Ein weiteres Ziel des PNK ist eine Antwort auf die Frage [DJ01]: Was ist ein Petrinetz? Es gibt einige Ansätze, die diese Frage mathematisch beantworten [EJPR01] (siehe auch den Ansatz des Petrinetz-Hyperwürfels in Kap. 3). Der PNK gibt eine software-technische Antwort, indem er eine Schnittstelle zum Zugriff auf ein Petrinetz und zur Definition von Petrinetz-Typen bereitstellt.

Zusammengefasst ergeben sich die folgenden Ziele und Prinzipien für die Entwicklung des PNK:

Einfachheit Der PNK soll von Nutzern verwendet werden, die keine Experten der Software-Technik sind. Es soll ohne Programmierung möglich sein, ein Petrinetz-Werkzeug aus Standardmodulen bzw. Anwendungsmodulen zusammenzusetzen. Ein Anwendungsmodul oder einen Petrinetz-Typ zu implementieren, darf nicht schwieriger sein als ohne die Verwendung des PNK.

Modularität Die Wiederverwendung von Anwendungsmodulen und Teilen von Petrinetz-Typen soll möglich sein. Für die Definition neuer Petrinetz-Typen soll es möglich sein, aus einer Liste von vordefinierten Merkmalen auszuwählen.

Flexibilität Der PNK soll genügend flexibel sein, um neue Funktionalität zu implementieren und neue Petrinetz-Typen zu definieren. Der PNK soll die Sichtweise eines Petrinetz-Experten auf ein Petrinetz wiedergeben und so die Implementierung mathematischer Konzepte als Anwendungsmodule unterstützen. Außerdem muss es möglich sein, Standardmodule gegen effizientere bzw. passendere Module auszutauschen.

Eindeutigkeit Der PNK soll jedes Petrinetz aus einer software-technischen Sicht wider-

spiegeln. Alle Objekte und Funktionen sollen eindeutig als Petrinetz-Objekte bzw. Funktionen über Petrinetzen interpretiert werden.

Konsistenz Der PNK muss die Struktur (den Graphen) eines von ihm verwalteten Netzes konsistent halten. Das heißt beispielsweise, es darf in einem Netz nur Kanten *zwischen* Knoten geben.

5.2 Konzepte im PNK

Wie die Petrinetz-Beschreibungssprache PNML (siehe Kap. 4) verfolgt der PNK ein einheitliches Labelkonzept. Im Gegensatz zu dem statischen Konzept von PNML ist das Labelkonzept im PNK um einen dynamischen Anteil erweitert (näheres dazu in Abschn. 5.2.3). Zu unterscheiden sind der konventionelle Ansatz und der Ansatz des Petrinetz-Hyperwürfels.

Der konventionelle Ansatz überlässt dem Nutzer des PNK, ein Label im Kontext eines Petrinetz-Typs zu behandeln. Es werden lediglich wenige Methoden für Label-Klassen gefordert. Der Ansatz nach dem Petrinetz-Hyperwürfel erfordert von jeder Label-Klasse die Implementierung bestimmter semantischer Aspekte, die ein Label für die parametrisierte Schaltregel zu erfüllen hat. Näheres dazu wird in Abschn. 5.4 erläutert.

5.2.1 Architektur

Der PNK besteht aus dem eigentlichen Kern, der Anwendungssteuerung und einer Bibliothek von Anwendungsmodulen und Petrinetz-Typen. Der Kern wird mit einem Petrinetz-Typ als Parameter instanziiert, so dass der Kern ein Netz des angegebenen Typs darstellt.

Abbildung 5.5 zeigt die Komponenten des PNK, wie sie sich einem PNK-Nutzer präsentieren. Er legt fest, mit welchen Petrinetz-Typen und welchen Anwendungsmodulen bzw. Dateiformaten ein PNK-Werkzeug umgehen soll. Dabei bezieht der PNK-Nutzer Anwendungsmodule jeweils auf eine Teilmenge der vorgesehenen Petrinetz-Typen.

Als Beispiel betrachten wir Abb. 5.6, die ein einfaches PNK-Werkzeug mit dem Namen SimpleTool darstellt. Dieses Werkzeug bearbeitet S/T-Netze und B/E-Netze. Es besteht aus einem Editor, der Netze beider Typen bearbeiten kann und einem Simulator für S/T-Netze. Das Dateiformat PNML wird für Netze beider Typen verwendet. S/T-Netze können zusätzlich auch im Dateiformat net geladen und gespeichert werden.

Ein PNK-Werkzeug ordnet auf dieser Basis, wie vom PNK-Nutzer gewollt, jedem Netz die entsprechenden Anwendungsmodule zu. Wir können mit Abb. 5.7 einen anderen Blickwinkel auf ein PNK-Werkzeug einnehmen. Danach ist jedem Netz, das als Netz eines bestimmten Typs instanziiert wurde, eine Reihe von Anwendungsmodulen

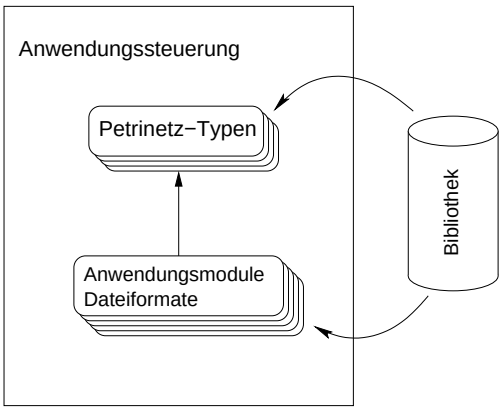


Abbildung 5.5: PNK als Entwicklungsumgebung

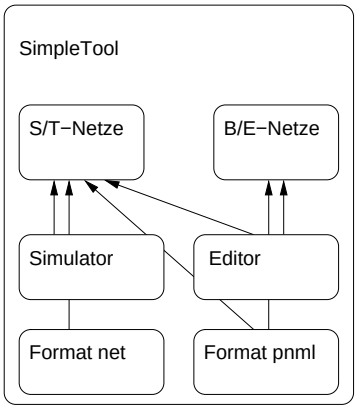


Abbildung 5.6: Beispiel eines PNK-Werkzeuges

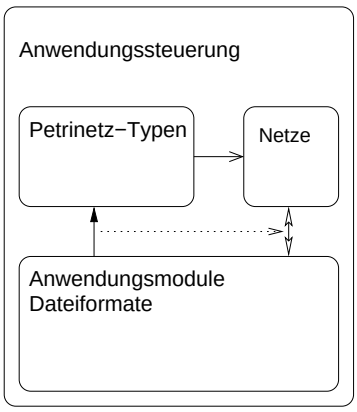


Abbildung 5.7: Funktionsweise eines PNK-Werkzeuges

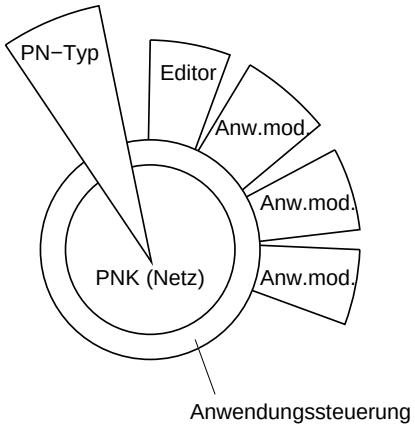


Abbildung 5.8: Sicht auf ein Netz im PNK

zugeordnet, die die Anwendungssteuerung aus den Anwendungsmodulen ermittelt, die auf den Typ des Netzes verweisen. Netze können dabei jeweils von einer eigenen Instanz eines Anwendungsmoduls benutzt werden, oder die Instanz eines Anwendungsmoduls bearbeitet gleichzeitig mehrere Netze. Sie ist also ein Anwendungsmodul mehrerer Netze.

Schließlich stellen wir uns die Sicht auf ein Netz in einem PNK-Werkzeug wie in Abb. 5.8 vor. Auf ein Netz, das mit einem Parameter für den Petrinetz-Typen instanziiert wurde, können verschiedene Anwendungsmodule, vermittelt durch die Anwendungssteuerung, zugreifen.

5.2.2 Kern

Jedes Petrinetz-Werkzeug muss in der Lage sein, einen Petrinetz-Graphen zu verwalten. Das heißt, entsprechend der Def. 3.1.2 bzw. 3.1.1 müssen Stellen, Transitionen, Kanten und ihre Beziehungen zueinander im Werkzeug dargestellt sein. Wir bezeichnen dies als die Standardfunktionalität eines Petrinetz-Werkzeuges. Außerdem verwaltet jedes Petrinetz-Werkzeug eine entsprechende Menge von Labeln (siehe Def. 3.3.1) und ihre Werte. Im Werkzeug muss klar sein, welches Netzelement welche Label mit welchen Werten hat. Das Netz selbst kann ebenfalls Label haben, z. B. die Menge der im Netz verwendeten Variablen.

Im PNK hat jedes gleichartige Netzelement die gleichen Label. Das heißt, ein Label ist nach Def. 3.3.1 eine totale Abbildung. Dies bedeutet keine Einschränkung für Label, da jedes Label einen Standardwert vereinbart, der von den Werten anderer Label abhängig sein darf.

5.2.3 Petrinetz-Typ

Der PNK unterstützt zwei verschiedene Konzepte zur Beschreibung eines Petrinetz-Typs – einen konventionellen Ansatz und den Ansatz des Petrinetz-Hyperwürfels. Unabhängig vom konkreten Konzept präsentiert der PNK seinem Nutzer eine einheitliche Schnittstelle für den Petrinetz-Typ.

Konventioneller Ansatz

Nach dem konventionellen Ansatz besteht ein Petrinetz-Typ aus einer beliebigen Menge von Labeln. Jedes Label ist dem Netz oder einem Netzelement zugeordnet. Entsprechend der Def. 3.3.1 ist ein Label eine totale Abbildung gleichartiger Netzelemente eines Netzes in eine Menge von Werten. Ein einzelnes Label verwaltet sein Netzelement, seinen Wert und verschiedene Operationen, die auf den Wert bezogen sind. Mit diesem flexiblen Ansatz können alle Label von Petrinetzen beschrieben werden.

Ein Label wird häufig nicht nur in Petrinetzen eines einzigen Typs verwendet. Beispielsweise benötigt jeder Petrinetz-Typ, der in seinen Netzen Markierungen zulässt, wie sie gewöhnlicherweise in S/T-Netzen vorkommen, ein Label zur Beschreibung dieser Markierung. Das heißt, alle derartigen Petrinetz-Typen teilen dieselbe Beschreibung. Damit muss ein Label nur einmal implementiert werden und steht dann für einen beliebigen Petrinetz-Typ zur Verfügung.

Der Nachteil dieses Ansatzes liegt darin, dass die Kontrolle, welche Label zu einem Petrinetz-Typ kombiniert werden, vollständig beim PNK-Nutzer liegt. Er hat dafür zu sorgen, dass voneinander abhängige Label richtig implementiert sind.

Nach diesem Ansatz ist die Schaltregel ein Label des Netzes. Sie hat einen konstanten Wert und implementiert verschiedene Operationen, z. B. zur Berechnung einer aktivierten Transition und zum Schalten eines Schrittes. Die Schaltregel ist ein Beispiel für ein Label, das das Vorhandensein bestimmter anderer Label, wie hier die Markierung einer Stelle, erfordert.

Ansatz nach dem Petrinetz-Hyperwürfel

Entsprechend der Ausführungen im Abschn. 3.6 besteht ein Petrinetz-Typ aus einer Markenmenge, einer Markierungsstruktur, einer Menge von Kantentypen, einer Menge von Labeluniversen und einer parametrisierten Schaltregel. Die Parameter der Schaltregel sind ein Schrittparameter, ein Abhängigkeitsparameter für Schritte und ein Auswahlparameter zur Definition einer Halbordnungsrelation über Schritten. Ein Netz als Instanz eines Petrinetz-Typs enthält somit neben dem Petrinetz-Graphen und den Labeln (auch den aus den Dimensionen des Petrinetz-Hyperwürfels abgeleiteten Labeln) einen Verweis auf die Instanz einer parametrisierten Schaltregel entsprechend Abschn. 3.5.

Der Vorteil dieses parametrisierten Ansatzes liegt auf der Hand: Die Dimensionen des Petrinetz-Hyperwürfels werden soweit wie möglich unabhängig voneinander implementiert. Die meisten Parameter für die Dimensionen können, bis auf besonders exotische Exemplare, ohne Kenntnis eines bestimmten Petrinetz-Typs entwickelt werden. Für Label gilt das gleiche wie für den konventionellen Ansatz. Lediglich die Menge der Marken muss für spezielle Varianten ein geeignetes Datentypkonzept implementieren. Mit entsprechenden Werkzeugen für diesen Zweck kann hier eine weitere Erleichterung geschaffen werden. Dies liegt jedoch außerhalb des Gegenstandes der vorliegenden Arbeit.

5.2.4 Anwendungsmodule

Die Petrinetz-Typ-Schnittstelle des PNK erlaubt es, verschiedene Versionen von Petrinetzen zu definieren. Wenn wir einen Petrinetz-Typ definiert haben, ist der PNK in der Lage, Petrinetze dieses Typs zu verwalten. Ein *Anwendungsmodul* des PNK ist nun die Software, die auf ein Netz zugreift, das vom PNK verwaltet wird. Zu diesem Zweck stellt der PNK eine Schnittstelle zur Verfügung, die es einem Anwendungsmodul erlaubt, die Netzstruktur und die vom Petrinetz-Typ definierten Label zu erzeugen, zu verändern und zu löschen.

Ein Anwendungsmodul nutzt die vom PNK bereitgestellten Schnittstellen zur Analyse eines Netzes, zur Veränderung des Netzes, zur Visualisierung von Analyseresultaten u. ä. Im Allgemeinen besteht ein PNK-Werkzeug aus mehreren derartigen Anwendungsmodulen. Auch solche Standardanwendungen wie ein Petrinetz-Editor oder Lade- und Speicheroperationen sind in unserer Betrachtungsweise Anwendungsmodule. Das erleichtert es beispielsweise, den grafischen Editor des PNK unabhängig von diesem zu entwickeln oder die Lade- und Speicheroperationen durch andere für ein anderes Dateiformat zu ersetzen bzw. zu ergänzen. Für den PNK wurde ein grafischer Editor implementiert, der zudem jedes Netz jeden Typs anzeigen und verändern kann, um den PNK-Nutzer gerade auch von dieser Arbeit zu entlasten. Ein PNK-Nutzer kann aber auch einen eigenen Editor implementieren, ohne die restlichen Module des PNK verändern bzw. ersetzen zu müssen.

5.2.5 Anwendungssteuerung

Die *Anwendungssteuerung* des PNK verwaltet Anwendungsmodule, Petrinetz-Typen und Netze. Sie koordiniert damit ein PNK-Werkzeug. Der Anwendungssteuerung sind alle aktuell geladenen Petrinetze eines Werkzeuges mitsamt den mit ihnen verbundenen Anwendungsmodulen bekannt. Damit kontrolliert sie insbesondere, dass Veränderungen eines Anwendungsmoduls in einem Netz an alle anderen Anwendungsmodule, die für dieses Netz registriert sind, weitergeleitet werden.

Außerdem verwaltet die Anwendungssteuerung die Interaktion zwischen einem Anwendungsmodul und dem Endnutzer. Ein Anwendungsmodul löst zu einem Netz eine Interaktion mit dem Endnutzer aus ohne das Anwendungsmodul zu kennen, das die konkrete Aktion realisiert. Vielmehr wird die Anwendungssteuerung mit der Interaktion beauftragt, die dann ein Anwendungsmodul auswählt und mit der Interaktion beauftragt, das dazu in der Lage ist.

Die Anwendungssteuerung präsentiert sich mit einer grafischen Benutzerschnittstelle, über die Netze erzeugt und Anwendungsmodule gestartet werden.

5.2.6 Ein PNK-Werkzeug

Ein *PNK-Werkzeug* wird installiert, indem der Anwendungssteuerung des PNK mitgeteilt wird, von welchem Typ die zu verwaltenden Netze sind und welche Anwendungsmodule jeweils instanziiert werden können. Ein PNK-Werkzeug besteht typischerweise aus einem Editormodul zur Visualisierung und Veränderung eines Netzes, einem Lade- und Speichermodul sowie wenigstens einem weiteren speziellen Anwendungsmodul.

5.3 Implementierung des PNK

In diesem Abschnitt wird die Implementierung des PNK beschrieben.

5.3.1 Design

Der PNK ist mit der objektorientierten Programmiersprache Java implementiert (siehe Abschn. 5.1.3). Der objektorientierte Ansatz kommt unserer Betrachtungsweise von Petrinetz-Graphen, ihren Elementen und den Labeln sehr entgegen. Für das Netz, jedes Netzelement, die Label und die Anwendungsmodule werden Klassen implementiert.

Petrinetz-Typen und PNK-Werkzeuge kombinieren lediglich Label und Netzelemente bzw. Petrinetz-Typen und Anwendungsmodule. Es ist daher nicht notwendig, für diesen eher deskriptiven Anteil der Entwicklungsarbeit eines PNK-Nutzers die Verwendung einer Programmiersprache vorzuschreiben. Stattdessen wird ein Petrinetz-Typ und ein PNK-Werkzeug mit Hilfe je einer XML-Datei konfiguriert.

Der PNK hat zwei allgemeine Anwendungsmodule, die für Petrinetze jeden Typs eingesetzt werden können. Sie erweitern den PNK um eine sinnvolle auf ein Petrinetz bezogene Basisfunktionalität, die ein jedes Petrinetz-Werkzeug benötigen kann: ein grafischer Editor für Petrinetze und ein Lade- und Speichermodul. Letzteres wird durch die Petrinetz-Beschreibungssprache PNML zur Verfügung gestellt (siehe Kap. 4).

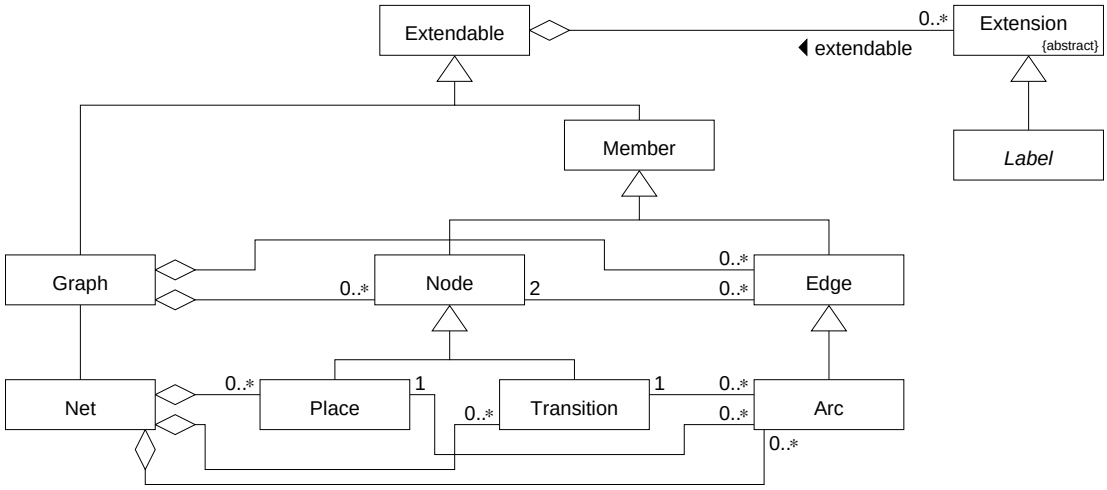


Abbildung 5.9: Klassendiagramm (UML) der Kernkomponente des PNK

5.3.2 Kernkomponente

Die Kernkomponente des PNK implementiert einen Petrinetz-Graphen und seine Netzelemente. Sie besteht aus den Klassen `Net`, `Place`, `Transition` und `Arc` sowie ihren jeweiligen Superklassen. Abbildung 5.9 zeigt das Klassendiagramm der Kernkomponente nach UML (siehe Abschn. 5.1.2).

Jede von der Klasse `Extendable` abgeleitete Klasse kann eine beliebige Anzahl von Label-Klassen aggregieren, die sämtlich von `Extension` abgeleitet sind. Mit einer Label-Klasse wird ein Label für Netze bzw. Netzelemente implementiert.

Als einzige der in Abb. 5.9 aufgeführten Klassen sind *Label*-Klassen dazu gedacht, von einem PNK-Nutzer implementiert zu werden. Alle anderen stellen eine endgültige Implementation dar. Sie werden über ihre Anwendungsschnittstelle (*application programming interface*; API) bedient. Die API der Kernkomponente enthält zahlreiche Methoden zur Verwaltung von Petrinetz-Elementen in Petrinetz-Werkzeugen, z. B. Anlegen, Ändern, Abfragen und Löschen. Im Allgemeinen wird ein PNK-Nutzer vor allem die Abfragemethoden der Kernkomponente benutzen. Beispielsweise liefert die Methode `net.getPlaces()` eine Liste aller Stellen des Netzes `net`, wenn `net` eine Instanz der Klasse `Net` des PNK ist.

Jedes Netzelement bzw. das Netz bietet Zugang zu seinen Labeln über die Methode `getExtension(string)`, wobei `string` den Namen bestimmt, unter dem das Label dem jeweiligen Netzelement bekannt ist. Der Name des Labels wird vom Petrinetz-Typ vereinbart, der den Namen mit der Implementation des Labels verbindet. Ein Label muss, wie in Abb. 5.9 gezeigt, von der PNK-Klasse `Extension` abgeleitet sein.

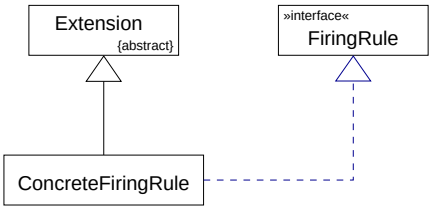


Abbildung 5.10: Klassendiagramm (UML) der Schaltregel

5.3.3 Deklaration eines Petrinetz-Typs

Ein Petrinetz-Typ ist, dem konventionellen Ansatz zufolge, eine Anzahl von Labeln, die jeweils einem Petrinetz-Element bzw. dem Netz zugeordnet sind und eine Schaltregel. Auch die Schaltregel ist ein Label, genauer ein unveränderliches Label aller Netze des Typs. Die Schaltregel eines Petrinetz-Typs implementiert die Schnittstelle `FiringRule` des PNK. Abb. 5.10 zeigt das Klassendiagramm eines Beispiels für eine konkrete Schaltregel (`ConcreteFiringRule`). Sie ist als Label des Netzes von der PNK-Klasse `Extension` abgeleitet und implementiert die Schnittstelle `FiringRule`.

Sowohl die Label für Petrinetz-Elemente bzw. das Netz als auch die Schaltregel des Petrinetz-Typs werden so weit wie möglich unabhängig und unter Nutzung der Schnittstellen des PNK implementiert. Das heißt dann, dass die Herstellung eines Petrinetz-Typs eher deklarativen Charakter hat und vom PNK-Nutzer nicht programmiert werden muss. Für die Deklaration eines Petrinetz-Typs verwenden wir XML-Dateien[1].

Programm 5.1 ist ein Beispiel einer Deklarationsdatei für einen Petrinetz-Typ. Es beginnt mit XML bezogenen Anweisungen wie der XML-Versionsnummer, dem Kodierschema der folgenden Zeichen sowie der Angabe der Grammatik, der die Datei folgt. Die Grammatik ist eine DTD (in diesem Fall *"netTypeSpecification.dtd"*).

Ein Petrinetz-Typ wird mit `netTypeSpecification` deklariert. Der Netztyp erhält einen Namen (`name`), in diesem Fall *"PTNet"* für Stellen-Transitionen-Netze. Dieser Name ist identisch mit dem Schlüsselnamen der entsprechenden PNTD (siehe Abschn. 4.3.3) und sollte auch für andere Dateiformate entsprechend benutzt werden. Der Petrinetz-Typ ist durch die Methode `getSpecification()` aus der API des PNK referenzierbar (siehe die Dokumentation der API [PNK00]). Jede Implementation eines Petrinetz-Elementes des Netzes wird innerhalb des XML-Elementes `netTypeSpecification` höchstens einmal mit dem XML-Element `extendable` aufgeführt. In seinem XML-Attribut `class` wird auf die Implementation des Netzelementes verwiesen. Das Argument ist ein Klassenpfad[2] nach Java-Konvention. Die Implementation des Netzelementes muss nicht notwendigerweise die Standardimplementation des PNK

[1] Zu XML siehe unsere Ausführungen in Abschn. 4.1.

[2] Der Klassenpfad wird in diesem und den folgenden Beispielen zuweilen Sinn wahrend gekürzt. Der vollständige Klassenpfad zum PNK lautet *"de.huberlin.informatik.pnk"*.

Programm 5.1: Deklaration eines Petrinetz-Typs des PNK

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE netTypeSpecification SYSTEM "netTypeSpecification.dtd">
<netTypeSpecification name="PTNet">
  <extendable class="pnk.kernel.Place">
    <extension name="marking"
               class="pnk.netElementExtensions.llNet.NaturalNumber"/>
    <extension name="initialMarking"
               class="pnk.netElementExtensions.llNet.NaturalNumber"/>
  </extendable>
  <extendable class="pnk.kernel.Arc">
    <extension name="inscription"
               class="pnk.netElementExtensions.llNet.NaturalNumber1"/>
  </extendable>
  <extendable class="pnk.kernel.Transition">
  </extendable>
</netTypeSpecification>
```

sein. Ein PNK-Nutzer kann durchaus eigene Implementationen von Netzelementen, z. B. für ganz exotische Petrinetz-Typen, verwenden. Im Allgemeinen ist dies nicht nötig; für die meisten Zwecke genügt die Implementation der Netzelemente im PNK. Die Implementation des Netzes muss nur referenziert werden, wenn das Netz Label enthält.

Nun wird jedes Label eines Netzelementes innerhalb der Deklaration des Netzelementes mit dem XML-Element `extension` vereinbart. Das Label hat einen Namen (`name`), der wiederum mit dem Labelnamen der Petrinetz-Typdefinition (PNTD) korrespondiert. Mit dem XML-Attribut `class` verweisen wir auf die Java-Klasse, die das Label implementiert. In dem Beispiel haben Stellen des Netzes zwei Label. Das eine Label speichert die initiale Markierung (*"initialMarking"*) und das andere die aktuelle Markierung (*"marking"*). Beide Label werden durch die gleiche Klasse implementiert. Außerdem haben die Kanten der Netze des hier vereinbarten Petrinetz-Typs Kanteninschriften (*"inscription"*). Alle Label werden hier als natürliche Zahlen aufgefasst und implementieren die entsprechenden Schnittstellen für Markierungen und Kanteninschriften. Die Implementationen der Klassen `NaturalNumber` für Markierungen und `NaturalNumber1` für Kanteninschriften unterscheiden sich lediglich in ihrem Standardwert. Der Standardwert einer Markierung ist 0; der für eine Kanteninschrift 1.

Der Name eines deklarierten Labels ist wichtig für Anwendungsmodule und die Implementation anderer Label. Eine einfache Schaltregel beispielsweise wird ein Label

mit dem Namen *"marking"* voraussetzen, um die Veränderungen auf einem solchen Label beim Schalten einer Transition zu implementieren.

Der PNK ergänzt zu jedem Netz und zu jedem Knoten ein Label *"name"*, das einen Bezeichner des Netzes bzw. des Knotens aufnimmt. Dieses Label ist das einzige Standardlabel des PNK. Die Schaltregel wird als Label des Netzes implementiert und muss, wenn gewünscht, auch als Label des Netzes deklariert werden. Eine Schaltregel implementiert die Schnittstelle `FiringRule` des PNK.

Der Ansatz der Deklaration eines Petrinetz-Typs des PNK ist vergleichbar einer Petrinetz-Typdefinition (PNTD) der Petrinetz-Beschreibungssprache PNML. Die Label, die in einer PNTD deklariert werden, enthalten im Gegensatz zur Petrinetz-Typdeklaration des PNK keine Operationen sondern lediglich eine Beschreibung ihrer zum Austausch fähigen Repräsentation.

5.3.4 Anwendungsmodule

Ein Anwendungsmodul des PNK wird implementiert, indem eine Java-Klasse programmiert wird, die die API des PNK benutzt. Ein Anwendungsmodul muss, damit es von der Anwendungssteuerung des PNK erkannt wird, von der Klasse `MetaApplication` abgeleitet sein. In dieser umfangreichen Klasse werden einige Methoden implementiert, die für die Verwaltung als Anwendungsmodul wichtig sind, aber einem PNK-Nutzer nur wenig nutzen. Deshalb werden weitere Klassen von `MetaApplication` abgeleitet, die viele Anwendungsfälle für den PNK abdecken und wesentlich übersichtlicher sind, da sie jeweils auf ihren Anwendungsfall zugeschnitten sind.

Abbildung 5.11 zeigt einen Ausschnitt aus dem Klassendiagramm für Anwendungsmodule des PNK. Anwendungsmodule werden in drei verschiedene Klassen unterteilt, die im Folgenden näher erläutert werden. Abbildung 5.11 zeigt außerdem einige Anwendungsmodule des PNK.

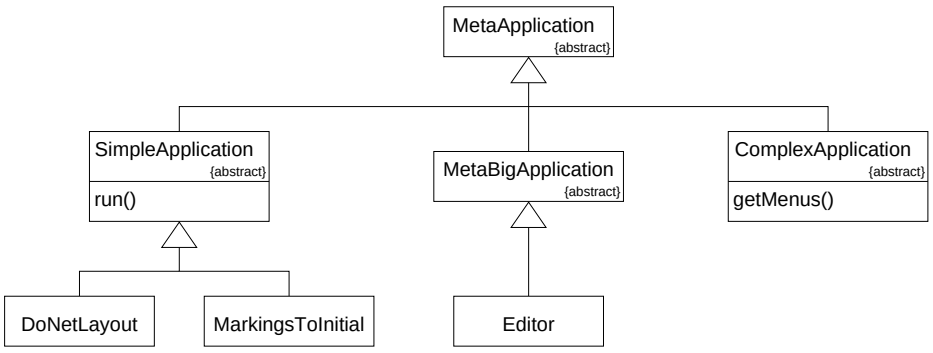


Abbildung 5.11: Klassendiagramm von Anwendungsmodulen des PNK

Programm 5.2: Ein einfaches Anwendungsmodul

```
package de.huberlin.informatik.pnk.app;

import de.huberlin.informatik.pnk.app.base.SimpleApplication;
import de.huberlin.informatik.pnk.appControl.ApplicationControl;
import de.huberlin.informatik.pnk.kernel.Net;

public class MarkingsToInitial extends SimpleApplication {

    public static String staticAppName = "MarkingsToInitial";
    public static boolean startAsThread = false;
    public static boolean startImmediate = true;

    public MarkingsToInitial(ApplicationControl ac) {
        super(ac);
    }

    public void run() {
        ((Net)net).resetMarkings(this);
        quitMe();
    }
}
```

Einfache Anwendungsfälle

Im einfachsten Fall möchte ein PNK-Nutzer einen bestimmten Algorithmus mit Hilfe des PNK implementieren. Dazu programmiert er die Methode `run()` einer Klasse, die von der abstrakten Klasse `SimpleApplication` abgeleitet ist. Die Anwendungssteuerung des PNK sorgt bei der Einbindung eines solchen Anwendungsmoduls dafür, dass ein Endnutzer das Anwendungsmodul über einen entsprechenden Menüeintrag starten kann.

Außerdem lässt es die Anwendungssteuerung zu, die Abarbeitung eines solchen einfachen Anwendungsmoduls „weich“ bzw. „hart“ zu beenden. Der Unterschied beider Beendigungsmethoden ist der folgende. Die „weiche“ Beendigung erfordert vom PNK-Anwendungsprogrammierer, dass sein Anwendungsmodul auf ein Signal der Anwendungssteuerung reagiert. Die „harte“ Beendigungsmethode zerstört das Objekt des Anwendungsmoduls.

Ein Beispiel für ein einfaches Anwendungsmodul ist das Progr. 5.2, das die Java-Klasse `MarkingsToInitial` zeigt. Dieses einfache Anwendungsmodul setzt die Markierung aller Stellen eines Netzes auf die Anfangsmarkierung. Es setzt voraus, dass der Petrinetz-Typ des Netzes für Stellen wenigstens zwei Label mit den Namen

"initialMarking" für die Anfangsmarkierung und *"marking"* für die aktuelle Markierung zulässt.

Das Beispiel in Progr. 5.2 beginnt mit der Deklaration des Java-Paketes, zu dem das Anwendungsmodul gehört, und dem Import von Klassen, die benutzt werden. Die Klasse `SimpleApplication` wird erweitert; ein Objekt der Klasse `ApplicationControl` ist die Anwendungssteuerung des PNK und ein Objekt der Klasse `Net` ist das Netz, das von dem Anwendungsmodul behandelt wird.

In Zeile 7 beginnt die Definition des Anwendungsmoduls `MarkingsToInitial` als eine von `SimpleApplication` abgeleitete Klasse. Zuerst werden Klassenvariablen der Superklasse gesetzt. Der Name des Anwendungsmoduls wird vereinbart. Das Anwendungsmodul soll nicht im Hintergrund, also nicht als Thread gestartet werden. Und das Anwendungsmodul soll sofort nach Aufruf bzw. Initialisierung gestartet werden. Im Gegensatz zu diesem Verhalten eines Anwendungsmoduls ist beispielsweise das Anwendungsmodul `DoNetLayout` nicht als sofort startendes Anwendungsmodul implementiert. Ein solches Anwendungsmodul startet erst nach besonderer Anforderung durch den Endnutzer.

Der Konstruktor eines Anwendungsmoduls (ab Zeile 13) wird immer mit einer Anwendungssteuerung als Parameter initialisiert. In dem Anwendungsmodul aus Progr. 5.2 wird die Anwendungssteuerung an den Konstruktor der Superklasse weitergereicht.

Die eigentliche Funktionalität des Anwendungsmoduls wird in der einzigen zu implementierenden Methode `run()` (ab Zeile 17) der Klasse implementiert. Diese Methode ruft die Methode `resetMarkings` eines Objektes der Klasse `Net`. Dazu wird die Objektvariable `net`, die das aktuelle Netz enthält und als Objekt der Klasse `Graph` vereinbart ist, in ein Objekt der Klasse `Net` umgewandelt. Die Methode `resetMarkings` des Netzes erwartet ein Objekt als Parameter, das der Auslöser der Aktion ist. Diese Methode setzt die Markierung jeder Stelle (Stellen-Label *"marking"*) auf den initialen Wert, wie er in dem Label *"initialMarking"* der Stelle enthalten ist. Schließlich wird das Anwendungsmodul mit der Methode `quitMe()` aus einer Superklasse dieser Klasse beendet.

Komplexe Anwendungsfälle

Komplexere Anwendungsmodule präsentieren dem Endnutzer ein eigenes Menü (bzw. mehrere davon). Das heißt, ein PNK-Nutzer implementiert für diesen Typ eines Anwendungsmoduls mehrere Methoden einer von `ComplexApplication` abgeleiteten Klasse entsprechender Funktionalität, die über die Methode `getMenus()` mit Menüeinträgen verbunden werden. Die Anwendungssteuerung sorgt für die Initialisierung des Anwendungsmoduls nach Anforderung durch den Endnutzer und präsentiert ihm die vom PNK-Nutzer implementierten Menüs und die Funktionalität ihrer Einträge.

Große Anwendungsmodule

Große oder besonders komplexe Anwendungsmodule benötigen eigene Fenster, Zeichenflächen und ähnliches sowie verschiedene Methoden zur Verwaltung und Kommunikation. Der Editor des PNK ist z. B. ein derartiges Anwendungsmodul. Solche sehr komplexen Anwendungsmodule werden von der Klasse `MetaBigApplication` abgeleitet. Für weitere Details sei an dieser Stelle auf die API-Dokumentation des PNK verwiesen [PNKoo].

Standardanwendungsmodule

Einige Anwendungsmodule sind für viele Petrinetz-Werkzeuge so wichtig, dass es berechtigt ist, sie als Standardanwendungsmodule zu bezeichnen. Ihre Aufgabe ist es, die auf den Petrinetz-Graphen bezogenen Kernkomponenten um solche Komponenten zu ergänzen, die es einem Endnutzer erleichtern, Petrinetze in das Werkzeug einzugeben. Das sind ein Editor zum Anlegen, Anzeigen, Ändern und Löschen von Netzelementen sowie ein Modul für Lade- und Speicheroperationen. Beide Komponenten sind so gestaltet, dass sie unabhängig vom Typ der Petrinetze sind. Von den Labeln ist bekannt, dass ihre Implementierung eine Zeichenkette liefern muss, die eine (externe) Repräsentation des Labelwertes darstellt. Diese Repräsentation kann sowohl in einem Editor angezeigt als auch in einer Datei abgespeichert werden. Die komplementäre Operation der Überführung einer externen Repräsentation in die interne Darstellung des Labels sichert den umgekehrten Weg der Eingabe von Labelwerten.

Der Editor des PNK[3] ist als Anwendungsmodul von der Klasse `MetaBigApplication` abgeleitet. Zusätzlich implementiert der Editor die Schnittstellen `NetObserver` und `ApplicationNetDialog`. Die Schnittstelle `NetObserver` deklariert ein Anwendungsmodul als Beobachter aller Netze, für die es gestartet wurde. Es wird damit von der Anwendungssteuerung des PNK über Veränderungen an beobachteten Netzen[4] informiert. Die Schnittstelle `ApplicationNetDialog` ermöglicht es anderen Anwendungsmodulen, mit einem Endnutzer über bestimmte Teile eines Netzes zu kommunizieren. Zum Beispiel präsentiert die Methode `selectObjects(objects)` einem Endnutzer (des Editors) eine Menge von Netzelementen (`objects`), von denen er, z. B. per Mausklick, genau eines auswählt, das an das aufrufende Anwendungsmodul zurückgegeben wird.

Wie zuvor erwähnt, werden auch Lade- und Speicheroperationen für Petrinetz-Dateien in Anwendungsmodulen implementiert. Ein solches Anwendungsmodul erbt von der abstrakten Klasse `InOut` des PNK. Die Lade- und Speicheroperationen sind gemeinsam in einer Klasse ausgeführt. Abbildung 5.12 zeigt den Entwurf des Lade- und Speichermoduls `InOut`. Da eine Netzdatei eines beliebigen Da-

[3] Das ist der Editor, der mit dem PNK mitgeliefert wird.

[4] Veränderungen am Netz werden tatsächlich vom Netz selbst an die Anwendungssteuerung gemeldet.

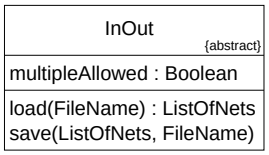


Abbildung 5.12: Klassendiagramm Lade- und Speichermodul

teiformates durchaus mehrere Netze aufnehmen kann, wird mit der Klassenvariablen `multipleAllowed` eingestellt, ob das Lade- und Speichermodul dies unterstützt. Dies ist vor allem für die Anwendungssteuerung des PNK nötig, um dem Endnutzer verschiedene Strategien zum Speichern von Netzen anzubieten. Die Methode `load` erwartet einen Dateinamen[5] als Parameter und liefert eine Liste von Netzen. Die Methode `save` erwartet als Parameter eine Liste von Netzen und einen Dateinamen[5].

Mit dem PNK wird ein Lade- und Speichermodul mitgeliefert, das das Einlesen und Abspeichern von PNML-Dateien implementiert. Da auch das PNML-Dateiformat mit Zeichenketten als Repräsentation von Labelwerten operiert, kann auf die vorhandene Schnittstelle für Label zurückgegriffen werden, um einen Labelwert zu speichern. PNML erlaubt es zudem, Netzelemente und Label mit grafischen und werkzeugspezifischen Informationen auszustatten (siehe Abschn. 4.2.1). Beim Speichern eines Netzes wird daher von der Anwendungssteuerung ein Signal an alle Anwendungsmodule gesendet, derartige Informationen in speziellen Datenstrukturen des Netzes abzulegen. Beim Laden werden diese speziellen Datenstrukturen genutzt, um grafische und werkzeugspezifische Informationen von PNML-Dateien abzulegen. Dort können sie von beliebigen Anwendungsmodulen gelesen werden.

Dieses Vorgehen sichert, dass Informationen in PNML-Dateien so weit wie möglich erhalten bleiben. Beispielsweise bleibt eine werkzeugspezifische Information zu einer bestimmten Stelle eines Netzes, die von einem beliebigen Petrinetz-Werkzeug erzeugt wurde, erhalten, auch wenn durch den PNK weitere Netzelemente hinzugefügt werden. Solange die betreffende Stelle nicht gelöscht wird, sind alle mit ihr verbundenen Daten reproduzierbar.

5.3.5 Anwendungssteuerung

Abbildung 5.13 zeigt die grafische Benutzerschnittstelle des PNK, wie sie sich einem Endnutzer präsentiert, wenn kein Netz geladen bzw. angelegt ist. Das Menü File enthält Funktionen zum Erzeugen eines Netzes eines bestimmten Typs, Lade- und Speicherfunktionen sowie den Ende-Knopf. Das Menü Net verwaltet später die Anwendungsmodule zu einem Netz und ist beim Start der Anwendungssteuerung inaktiv. Sobald ein Netz geladen bzw. erzeugt wird, startet die Anwendungssteuerung

[5] Tatsächlich wird ein *Uniform Resource Locator* (URL) erwartet.

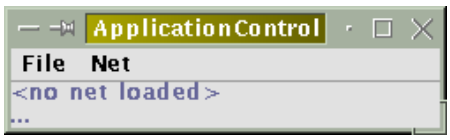


Abbildung 5.13: Anwendungssteuerung des PNK (grafische Benutzerschnittstelle)

Abbildung 5.14: Anwendungssteuerung mit aktivem Anwendungsmodul

die Standardanwendungsmodule eines PNK-Werkzeuges (siehe Abschn. 5.3.7). Ein Anwendungsmodul ergänzt die Menüs der Anwendungssteuerung mit eigenen Menüs. Ein Beispiel für eine Anwendungssteuerung mit einem aktiven Anwendungsmodul zeigt Abb. 5.14. Die Benutzeroberfläche der Anwendungssteuerung zeigt nun den Namen des Netzes, seinen Typ und das für dieses Netz aktive Anwendungsmodul (in diesem Fall den Editor, das Standardanwendungsmodul).

5.3.6 Kommunikation der Anwendungsmodule

Ein wichtiges Konzept der Anwendungssteuerung des PNK ist die Interaktion der Anwendungsmodule mit einem Endnutzer. Eine Interaktion wird in ein Objekt verpackt und der Anwendungssteuerung übergeben. Abbildung 5.15 zeigt das Klassendiagramm für diese Interaktionsobjekte. Jedes Interaktionsobjekt erbt von der abstrakten Klasse `MetaActionObject`. Diese Klasse implementiert die Methode `invokeAction`, die von der Anwendungssteuerung aufgerufen wird. Jede von `MetaActionObject` abgeleitete Klasse implementiert die Methoden `checkInterface` und `performAction`. Die erstere Methode überprüft, ob das übergebene Anwendungsmodul eine bestimmte Schnittstelle implementiert. Die letztere Methode setzt mit Hilfe des übergebenen Anwendungsmoduls die gewünschte Interaktion um.

Die in Abb. 5.15 dargestellten Interaktionsobjekte implementieren z. B. die folgenden Interaktionen. Ein Objekt der Klasse `AnnotateObjectsAction` beschriftet Netzobjekte in einem Anwendungsmodul, das die Schnittstelle `ApplicationNetDialog` implementiert. Ein Objekt der Klasse `SelectObjectAction` lässt den Endnutzer ein Netzobjekt aus einer Liste von Netzobjekten auswählen.

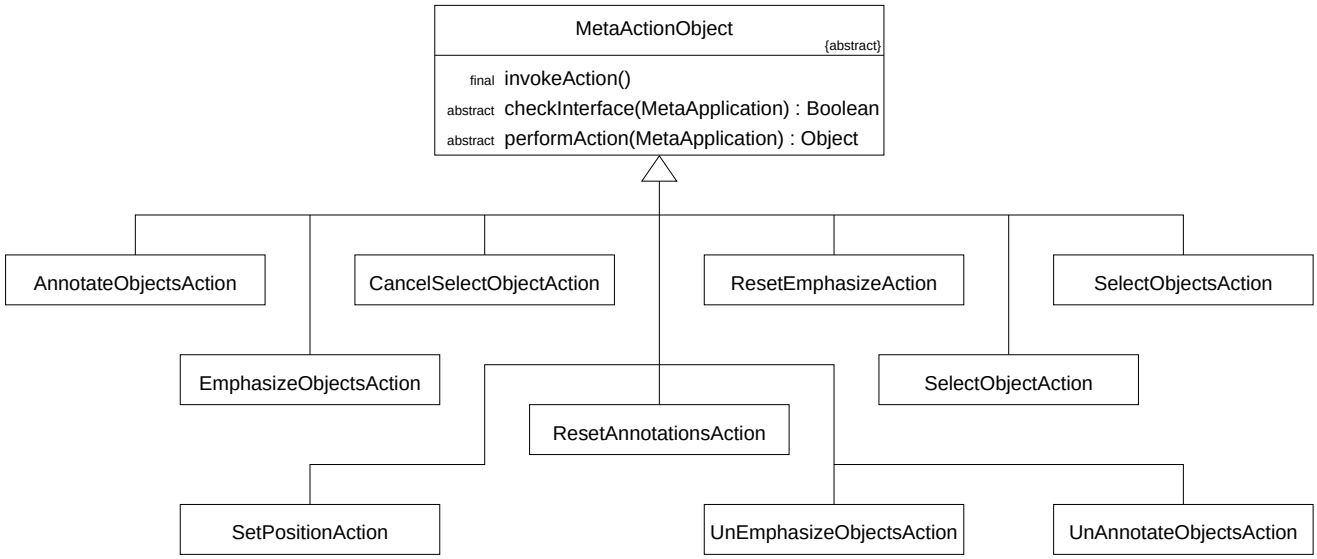


Abbildung 5.15: Klassendiagramm der Interaktionsobjekte

5.3.7 PNK-Werkzeug

Wie in Abschn. 5.2.6 bereits ausgeführt, ist die Zusammenstellung eines PNK-Werkzeuges eine deklarierende Aufgabe: Anwendungsmodule werden zu Petrinetz-Typen in Beziehung gesetzt. Ein PNK-Werkzeugentwickler beschreibt sein PNK-Werkzeug in einer XML-Datei, ähnlich wie für den Petrinetz-Typ in Abschn. 5.3.3.

Programm 5.3 zeigt die Beschreibung eines PNK-Werkzeuges (vgl. Abb. 5.7). Dieses Werkzeug ist ein grafischer Editor für Stellen/Transitions-Netze und Bedingungs/Ereignis-Netze. Für S/T-Netze stellt dieses Werkzeug auch noch einen Markenspielsimulator zur Verfügung und liest sowohl PNML-Dateien als auch Dateien einer älteren Version des PNK [KW99, Sch99]. Dagegen werden B/E-Netze nur als PNML-Dateien gelesen bzw. gespeichert. Sie können mit diesem Beispielwerkzeug nicht simuliert werden.

Die Beschreibung eines PNK-Werkzeuges beginnt mit XML-spezifischen Angaben zur XML-Version und zum Kodierschema. In Zeile 2 von Progr. 5.3 wird auf die DTD der Datei verwiesen. Das XML-Element `toolSpecification` ist das Wurzelelement der Datei. Zunächst werden die Petrinetz-Typen vereinbart, für die das Werkzeug verwendet wird, mit je einem XML-Element `nettype`. Jeder Petrinetz-Typ erhält eine Identifizierung (`id`) und einen Verweis auf seine Deklaration (`typeSpecification`). Diese Deklaration ist eine XML-Datei wie sie in Abschn. 5.3.3 beschrieben wurde. Im nächsten Abschnitt der Werkzeugkonfiguration wird mit dem Schlüsselwort `application` jeweils ein Anwendungsmodul des PNK-Werkzeuges beschrieben. Jedes Anwendungsmodul

Programm 5.3: Konfiguration eines PNK-Werkzeuges

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE toolSpecification SYSTEM "toolSpecification.dtd">
<toolSpecification>
  <!-- net types -->
  <nettype id="n1" typeSpecification="file:PTNet.xml"/>
  <nettype id="n2" typeSpecification="file:CENet.xml"/>
  <!-- application modules -->
  <application id="a1" mainClass="pnk.editor.Editor" maxinstances="inf">
    <allowedNettypes>
      <ntref ref="n1"/>
      <ntref ref="n2"/>
    </allowedNettypes>
  </application>
  <application id="a2" mainClass="pnk.app.Simulator" maxinstances="1">
    <allowedNettypes>
      <ntref ref="n1"/>
    </allowedNettypes>
  </application>
  <!-- Input/Output -->
  <format id="pnml" ioClass="pnk.appControl.PnmlInOut">
    <allowedNettypes>
      <ntref ref="n1"/>
      <ntref ref="n2"/>
    </allowedNettypes>
  </format>
  <format id="net" ioClass="pnk.appControl.NetInOut">
    <allowedNettypes>
      <ntref ref="n1"/>
    </allowedNettypes>
  </format>
  <!-- default settings -->
  <standardNettype ref="n1"/>
  <standardApplication ref="a1"/>
  <standardFormat ref="pnml"/>
</toolSpecification>
```

hat eine eindeutige Identifizierung und verweist mit dem XML-Attribut `mainClass` auf seine Implementation. Die Zeichenkette des XML-Attributes `mainClass` ist ein Klassenpfad nach Java-Konvention. Außerdem wird für Anwendungsmodule angegeben, wie viele ihrer Instanzen in einem PNK-Werkzeug gestartet werden dürfen (`maxinstances`). Innerhalb des XML-Elementes `application` werden im XML-Element `allowedNettypes` diejenigen Petrinetz-Typen referenziert, für deren Netze das Anwendungsmodul gestartet werden darf. Das XML-Element `ntref` verweist dabei mit seinem Attribut `ref` auf je einen der oben deklarierten Petrinetz-Typen. Für das Beispielwerkzeug wurde ein Anwendungsmodul vereinbart, implementiert in *`"pnk.editor.Editor"`*, das mit Netzen beider deklarierter Petrinetz-Typen operiert; während das Anwendungsmodul des Markenspielsimulators (deklariert in den Zeile 14 bis 18) nur für S/T-Netze vereinbart wurde.

Ein Anwendungsmodul, das eine Lade- und eine Speicheroperation enthält, implementiert eine etwas andere Schnittstelle des PNK. Deshalb und aus Gründen der Bedienbarkeit werden diese Anwendungsmodule mit dem XML-Element `format` deklariert. Ansonsten ähnelt das XML-Element im Aufbau und Inhalt dem für andere Anwendungsmodule. Auch hier wird auf die Implementation des Dateiformates (mit `ioClass`) und die für dieses Format zulässigen Petrinetz-Typen (`ntref`) Bezug genommen. In dem Beispielwerkzeug wurden zwei Dateiformate deklariert. Beide Formate sind dazu gedacht S/T-Netze zu laden und zu speichern. Für B/E-Netze wird jedoch nur das PNML-Format verwendet.

Schließlich wird für das PNK-Werkzeug aus Progr. 5.3 vereinbart, welches Anwendungsmodul, welches Dateiformat und welcher Netztyp als vorausgewählter Standard angenommen werden soll. Das jeweilige XML-Element verwendet mit `ref` dazu den Referenzmechanismus von XML.

Ein PNK-Werkzeug wird gestartet, indem der PNK bzw. seine Anwendungssteuerung mit einer entsprechenden Werkzeugkonfiguration aufgerufen wird. Die Anwendungssteuerung des PNK liest eine derartige Werkzeugkonfiguration und verwaltet daraufhin Netztypen und Netze sowie Anwendungsmodule und ihre Beziehung zu Netztypen bzw. deren Netzen. Es instanziiert das Standardanwendungsmodul und das Standarddateiformat. Und es erzeugt die entsprechenden Menüs der jeweils aktiven Anwendungsmodule.

5.4 Der Petrinetz-Hyperwürfel in Anwendungsmodulen des PNK

In diesem Abschnitt wird das Konzept des Petrinetz-Hyperwürfels (siehe Kap. 3) validiert, indem ein parametrisierten Petrinetz-Typ des PNK entworfen wird. Dieser parametrisierte Petrinetz-Typ nimmt Parameter entsprechend der Dimensionen des Petrinetz-Hyperwürfels und der parametrisierten Schaltregel auf. Bereits für den Petrinetz-Würfel wurde mit Hilfe des PNK ein Werkzeug implementiert, das für einen

parametrisierten klassischen Petrinetz-Typ einen Markenspielsimulator implementiert [Web99].

Der wesentliche Nutzen des Petrinetz-Hyperwürfels ist, dass einige Label für Netze eines Petrinetz-Typs aus den Parametern für die Dimensionen des Petrinetz-Typs im Petrinetz-Hyperwürfel abgeleitet werden und dass die Schaltregel auf Grund allgemeiner Gesetzmäßigkeiten unabhängig von konkreten Petrinetz-Typen implementiert werden kann. Die Label, die aus Parametern für Dimensionen des Petrinetz-Hyperwürfels abgeleitet werden, sind die Markierung einer Stelle, die Kanteninschrift (Kantengewicht), der Kantentyp und die Schaltregel. Ein Petrinetz-Typ kann darüberhinaus weitere Label definieren, die in der Schaltregel eine Rolle spielen. Zu unterscheiden sind statische und dynamische Label. Statische Label haben konstante Werte und beeinflussen auf Grund ihrer Werte die verschiedenen Phasen der Schaltregel. Dynamische Label dagegen werden durch zusätzliche Schaltfunktionen verändert. Ein dynamisches Label kann auch wie ein statisches Label fungieren.

5.4.1 Markenmenge

Die erste Dimension des Petrinetz-Hyperwürfels beeinflusst die Label Markierung (von Stellen) und Kanteninschrift bzw. dessen Belegungsfunktion. Die Implementation dieser Dimension definiert, was eine Marke eines Netzes des entsprechenden Typs ist. Es wird also eine abstrakte Klasse `Token` implementiert, als Superklasse aller konkreten Implementationen für Marken. Darin definieren wir eine Schnittstelle. Diese Schnittstelle enthält Methoden zur Umwandlung der internen Repräsentation des Markenwertes in eine externe bzw. umgekehrt sowie eine Methode zum Vergleich mit einer anderen Marke.

Abbildung 5.16 zeigt den Entwurf der Klasse `Token`. Die Klasse wird durch eine Ausnahmebehandlung ergänzt für den Fall, dass eine übergebene Zeichenkette keine gültige Repräsentation einer Marke dieser Klasse ist. Von dieser Klasse werden alle konkreten Klassen für Marken abgeleitet. Es gibt eine Klasse für schwarze Marken und mehrere für *high level* Marken, die mit anderen Label-Klassen zusammenarbeiten, z. B. einem Label für Sorten.

Eine besondere Klasse für Marken ist die Klasse für Markenvariablen `VarToken`, die von der Klasse `Token` abgeleitet wird. Sie wird verwendet, um in Ausdrücken wie Kanteninschriften und Schaltnebenbedingungen für Transitionen die Marken auszudrücken, die erst von einer Belegung konkrete Marken erhalten (siehe Abschn. 3.5.1). Eine Markenvariable hat einen Datentyp (`DataType`), der die Werte für Marken beschreibt, die diese Variable repräsentieren kann. Die Methoden `setValue` der Klasse `VarToken` setzt den Wert der Variable auf eine bestimmte Marke. Die Methode `getValue` liefert den aktuellen Wert der Variable.

Damit wurde ein Rahmen und Beispiele zur Implementierung des ersten Parameters des Petrinetz-Hyperwürfels angegeben.

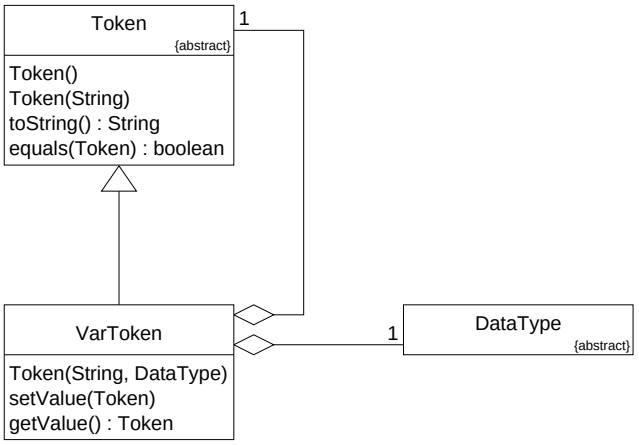


Abbildung 5.16: Entwurf der Klasse `Token`

5.4.2 Markierungsstruktur

Die Markierungsstruktur bestimmt, wie Marken zu Markierungen von Stellen zusammengesetzt sind. Wir entwerfen eine abstrakte Klasse `MarkingStructure`, von der konkrete Klassen für Markierungsstrukturen wie Menge, Multimenge und Sequenz abgeleitet werden. Da es in Java keine Mehrfachvererbung gibt, erbt diese Klasse von der PNK-Klasse `Marking`, um die Markierung einer Stelle bilden zu können. Ein Objekt einer solchen Klasse verwaltet die enthaltenen Objekte (Marken) entsprechend einer bestimmten Algebra. Es bildet also genau ein Element der Algebra ab.

Wir benötigen für eine solche Klasse Methoden zum Hinzufügen (`add`) und Entfernen (`sub`) anderer Instanzen derselben Klasse. Weiterhin benötigen wir eine Methode, die überprüft, ob die Struktur leer ist, d. h. ob sie dem neutralen Element gleich ist (`isEmpty`), und Methoden, die überprüfen, ob die oben genannten Operationen der Algebra zulässig sind oder nicht (`validAdd` und `validSub`). Von der Klasse `Marking` wird eine Methode `toString` geerbt, um die externe Repräsentation der Markierungsstruktur zu bilden. Sie baut auf der externen Repräsentation der in ihr enthaltenen Objekte (Marken) auf. Der Konstruktor der Klasse erzeugt entweder eine leere Markierungsstruktur oder eine Markierungsstruktur, die eine Instanz der Menge `Token` enthält, oder eine Markierungsstruktur, die aus einer externen Repräsentation abgeleitet wird.

Abbildung 5.17 zeigt den Entwurf der abstrakten Klasse `MarkingStructure`. Entsprechend der Ausführungen in Abschn. 3.2.2 benötigen wir davon abgeleitet wenigstens Implementationen für Menge, Multimenge, Sequenz und Stapel. Jede Markierungsstruktur enthält eine beliebige Anzahl von Marken. Wieviele es tatsächlich sein dürfen hängt von der konkreten Implementation der entsprechenden Klasse für die

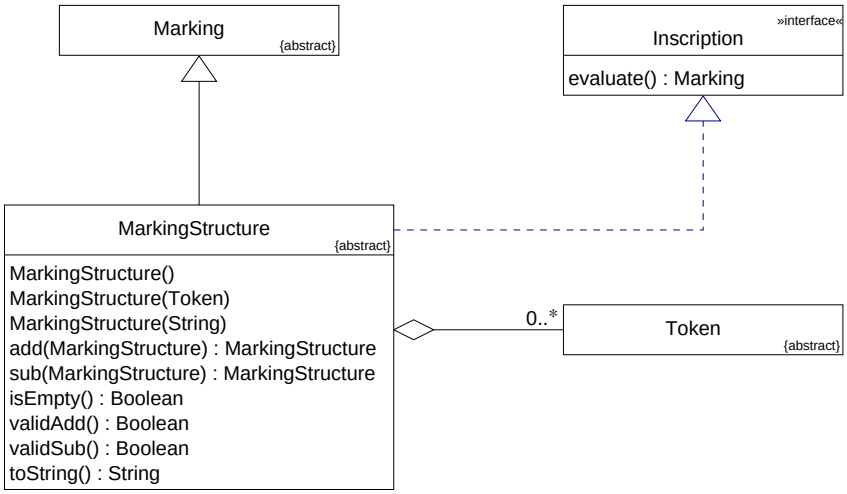


Abbildung 5.17: Entwurf der Klasse `MarkingStructure`

Markierungsstruktur ab. Beispielsweise nimmt die Klasse einer Menge als Markierungsstruktur nur ungleiche Marken als Elemente auf. Mit Hilfe der Methode `equals` aus der Klasse `Token` wird diese Bedingung überprüft. Die Klasse `MarkingStructure` wird durch eine Ausnahmebehandlung für unzulässige Additions- und Subtraktionsoperationen ergänzt.

Die Klasse `MarkingStructure` enthält neben den drei Konstruktoren die Methoden `add` und `sub` zum Hinzufügen bzw. Entfernen aller Elemente einer weiteren Markierungsstruktur, die als Parameter übergeben werden. Außerdem hat die Klasse Methoden zur Überprüfung, ob diese beiden Methoden zulässig sind, und eine Methode, die überprüft, ob eine Markierungsstruktur leer ist. Außerdem implementiert die Methode `toString` die externe Repräsentation eines Objektes der Klasse.

Eine Kanteninschrift ist einer Markierung in ihrer inneren Struktur ähnlich. Technisch stellt sie eine konstante Markierungsstruktur dar, wobei abstrakte Marken (Variablen) und Funktionen integriert sein können. Auf Grund dieser Ähnlichkeit implementiert die Markierungsstruktur auch die Schnittstelle für eine Kanteninschrift, so dass eine konkrete Klasse für Kanteninschriften von der Klasse `MarkingStructure` abgeleitet wird. Die Methode `evaluate` der Schnittstelle `Inscription` wird dabei so implementiert, dass aus jeder Marke der Markierungsstruktur, die eine Variable darstellt, eine konkrete Marke erzeugt wird, die der aktuellen Belegung entspricht.

5.4.3 Kantentypen

Ein Kantentyp muss zwei Funktionen implementieren; eine überprüft die Aktivierungsbedingung, die andere führt den Effekt aus. Entsprechend unserer Ausführungen in

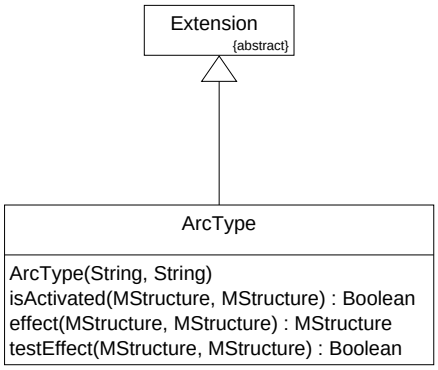


Abbildung 5.18: Klassenentwurf der Kantentypen

den Abschn. 3.4.2 und 3.4.3 operieren diese Funktionen auf Elementen der Markierungsstruktur.

Abbildung 5.18 zeigt den Klassenentwurf für den Kantentyp. Er wird als Label einer Kante implementiert und deshalb von der PNK-Klasse `Extension` abgeleitet. Die Klasse `ArcType` ist eine konkrete Klasse und implementiert den Kantentyp mit Hilfe von zwei Parametern wie in Tab. 3.1. Es wird eine weitere Methode eingeführt, die den Effekt auf Zulässigkeit überprüft, da nur die widersprüchlichen Kombinationen (PUT-INHIB, ADD) und (INHIB, SUB) *a priori* ausgeschlossen werden. Diese Methode kann auch verwendet werden, um die Zulässigkeit des kumulierten Effektes der parametrisierten Schaltregel zu überprüfen (vgl. Def. 3.5.7). Die Klasse `ArcType` wird durch eine Ausnahmebehandlung für nicht zulässige Effekte ergänzt.

Der Konstruktor der Klasse `ArcType` erwartet zwei Parameter entsprechend unserer Klassifikation in Tab. 3.1. Die Klasse hat drei Methoden. Jede dieser Methoden erwartet zwei Objekte der Klasse `MarkingStructure` als Parameter (in der Abb. 5.18 durch MStructure abgekürzt). Die Methode `isActivated` überprüft, ob die Aktivierungsbedingung erfüllt ist. Die Methode `testEffect` überprüft, ob der Effekt zulässig ist. Die Methode `effect` führt den Effekt aus und liefert die resultierende Markierungsstruktur.

5.4.4 Label

Nicht schaltrelevante Label des Petrinetz-Hyperwürfels werden wie sonst im PNK auch implementiert. Sie sind hier nicht weiter von Interesse. Zusätzlich gibt es schaltrelevante statische Label und dynamische Label. Diese benötigen, den Ausführungen in Abschn. 3.3 zu Folge, Methoden, die von der parametrisierten Schaltregel benutzt werden. Schaltrelevante statische Label haben zwei Methoden, die das Aktivierungsprädikat bzw. Effektprädikat implementieren. Dynamische Label haben zwei Methoden,

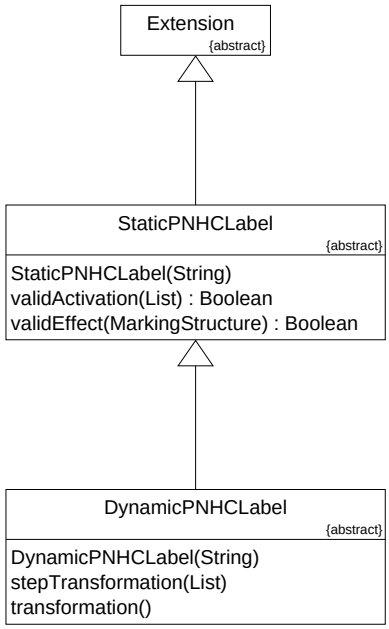


Abbildung 5.19: Klassenentwurf der Label des parametrisierten Petrinetz-Typs

die die Schrittüberführungsfunktion bzw. Nicht-Schrittüberführungsfunktion implementieren.

Abbildung 5.19 zeigt den Klassenentwurf der Label des parametrisierten Petrinetz-Typs. Die beiden Klassen `StaticPNHCLabel` und `DynamicPNHCLabel` werden nicht als Schnittstellen ausgeführt, da ihre Methoden die trivialen Fälle bereits implementieren. Da ein dynamisches Label auch ein statisches schaltrelevantes Label sein kann, erbt die Klasse `DynamicPNHCLabel` von `StaticPNHCLabel`.

Die Klasse `StaticPNHCLabel` implementiert eine Methode `validActivation`, die als Parameter eine Menge von Schaltmodi erwartet und überprüft, ob das Label mit der Menge der Schaltmodi aktiviert ist (siehe Abschn. 5.4.5). Außerdem hat die Klasse eine Methode `validEffect` , die überprüft, ob die als Parameter übergebene Markierungsstruktur im Sinne des Labels gültig ist. Die jeweilige triviale Implementierung der beiden Methoden liefert in jedem Falle den Wert `true`.

Die Klasse `DynamicPNHCLabel` implementiert zwei Überführungsfunktionen, die den Wert des Labels auf Grund eines Parameters, einem schaltenden Schritt, (`stepTransformation`) oder immer ändert (`transformation`). Die triviale Implementierung beider Methoden ändert den Labelwert nicht.

5.4.5 Schaltregelparameter

Die Schaltregel des Petrinetz-Hyperwürfels (siehe Abschn. 3.5) ist selbst parametrisiert. Die Parameter der Schaltregel sind der Schrittparameter, der Abhängigkeitsparameter und der Auswahlparameter für Schritte. Sie werden im Folgenden erläutert.

Schrittparameter

Der Schrittparameter bestimmt die Zusammensetzung eines Schrittes. Es wird eine Klasse implementiert, die aus einer Menge von Schaltmodi entsprechend des Parameterwertes eine Menge von Multimengen von Schaltmodi konstruiert.

Abbildung 5.20 zeigt den Entwurf der Klasse `StepStructure`. Ein Objekt dieser Klasse aggregiert eine Menge von (aktivierten) Schaltmodi. Im Allgemeinen wird diese Menge von der Schaltregel erzeugt. In Ableitungen dieser Klasse werden die konkreten Schrittparameter implementiert. Dazu wird die Methode `getNext` entsprechend implementiert. Diese Methode liefert eine Liste von Schaltmodi, die von der Schaltregel als Schrittkandidat behandelt wird und weiter analysiert wird. In der Abbildung 5.20 werden einige konkrete Implementierungen des Schrittparameters wiedergegeben. Die Klasse `StepStruct` implementiert einen Schrittkandidaten als Einermenge von Schaltmodi; die Klasse `StepStructSet` als Menge und die Klasse `StepStructMultiset` als Multimenge.

Abhängigkeitsparameter

Der Abhängigkeitsparameter bestimmt, ob eine Multimenge von Schaltmodi (Schrittkandidat) ein Schritt ist oder nicht. Es wird eine Klasse implementiert, die diese Überprüfung übernimmt. Der Abhängigkeitsparameter kann als lokaler Test eines Schrittes aufgefasst werden, denn es wird ausschließlich auf Grund der Schaltmodi des Schrittkandidaten entschieden, ob er ein Schritt ist.

Abbildung 5.21 zeigt den Entwurf der abstrakten Klasse `StepVerifier` und zwei konkrete Abhängigkeitsparameter. Die Klasse `StepVerifier` enthält eine abstrakte Me-

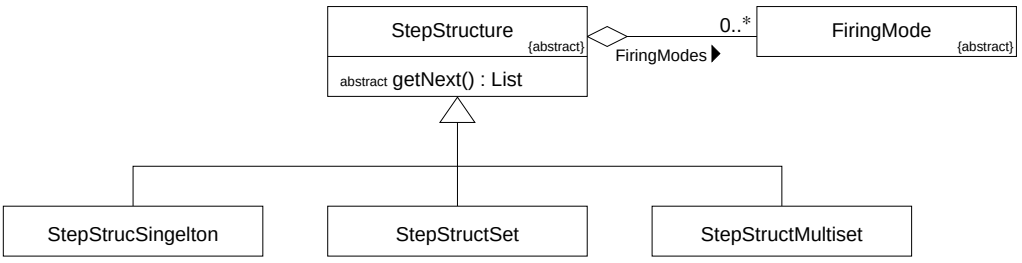


Abbildung 5.20: Klassenentwurf Schrittparameter

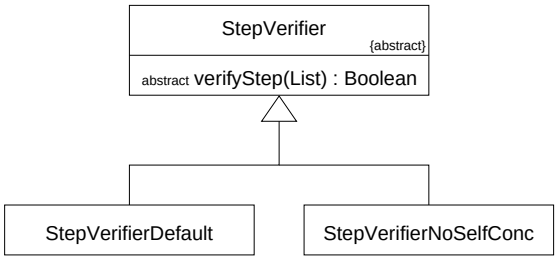


Abbildung 5.21: Klassenentwurf Abhängigkeitsparameter

thode `verifyStep`, die einen übergebenen Schrittkandidaten überprüft. Ein konkreter Abhängigkeitsparameter wird implementiert, indem diese Methode entsprechend implementiert wird. Beispielsweise liefert die Methode der Klasse `StepVerifierDefault` immer den Wahrheitswert `true`. Ein anderes Beispiel für einen Abhängigkeitsparameter ist die Klasse `StepVerifierNoSelfConc`, deren Methode `verifyStep` diejenigen Schrittkandidaten nicht als Schritte akzeptiert, deren Schaltmodi eine Transition mehrfach enthalten.

Auswahlparameter

Mit dem Auswahlparameter wird aus einer Menge von Schritten, die Teilmenge der aktivierten Schritte berechnet. Dazu werden Schritte entsprechend einer Halbordnungsrelation miteinander verglichen. Die maximalen Elemente dieser Halbordnungsrelation sind die aktivierten Schritte. Im Gegensatz zum Abhängigkeitsparameter müssen zuvor alle Schritte bekannt sein. Dieser Parameter kann also als globaler Test der Schritte aufgefasst werden.

Abbildung 5.22 zeigt den Klassenentwurf der Klasse `StepSelector` und zwei konkrete Auswahlparameter. Die abstrakte Methode `selectStep` wird in den abgeleiteten Klassen implementiert. Sie erwartet einen Parameter, der eine Menge von Schritten

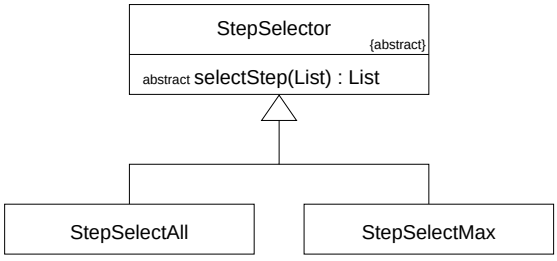


Abbildung 5.22: Klassenentwurf des Auswahlparameters

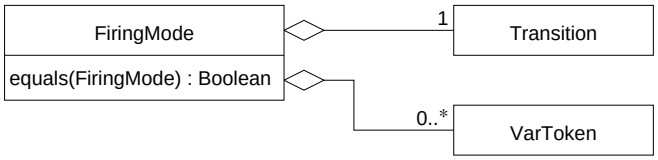


Abbildung 5.23: Klassenentwurf Schaltmodus

darstellt. Der Parameter ist also eine Menge von Multimengen von Schaltmodi. Aus dieser Menge von Schritten wird eine Teilmenge ausgewählt

Die Methode `selectStep` der Klasse `StepSelectAll`, die in Abb. 5.22 ebenfalls dargestellt wird, gibt den übergebenen Parameter einfach zurück; während dieselbe Methode der Klasse `StepSelectMax` nur die Schritte zurückgibt, die maximal viele Schaltmodi enthalten.

Die Schaltregelparameter werden von der nachfolgend zu erläuternden Schaltregel verwendet.

5.4.6 Schaltregel des Petrinetz-Hyperwürfels

Die Schaltregel des Petrinetz-Hyperwürfels implementiert die Schnittstelle `FiringRule` des PNK. Darin werden alle Phasen der Schaltregel implementiert. Zunächst werden alle aktivierten Schaltmodi berechnet. Dabei werden die Aktivierungsbedingungen der Kanten und der Label verwendet. Dann werden mit Hilfe des Schrittparameters und des Abhängigkeitsparameters (siehe Abschn. 5.4.5) aus den aktivierten Schaltmodi Schritte konstruiert. Aus dieser Menge der Schritte wird dann mit Hilfe des Auswahlparameters die Menge der tatsächlich aktivierten Schritte berechnet. Und schließlich wird ein aktivierter Schritt geschaltet, z. B. nachdem er von einem Endnutzer als zu schaltender Schritt ausgewählt wurde.

Dies erfordert je eine Datenstruktur für Schaltmodi und Schritte. Ein Schaltmodus setzt sich aus einer Transition und einer Belegung zusammen. Eine Belegung ist eine Menge von Markenvariablen `VarToken` mit aktuellen Werten. Abbildung 5.23 zeigt den Klassenentwurf des Schaltmodus. Ein Schaltmodus kann mit einem anderen Schaltmodus verglichen werden (Methode `equals`). Zwei Schaltmodi sind gleich, wenn sie die gleiche Transition enthalten und die gleichen Markenvariablen (`VarToken`) die gleichen Werte enthalten. Für einen Schritt verwenden wir als Datenstruktur eine Liste (vgl. Abb. 5.20).

5.4.7 Zusammenfassung

Ein parametrisierter Petrinetz-Typ des PNK besteht, wie zuvor erläutert, aus den folgenden sechs Parametern:

Programm 5.4: Ein leeres Netz eines parametrisierten Petrinetz-Typs (PNML)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<pnml>
  <net id="n1" type="PNHCube(BlackToken, MultiSet, extended, multi set of
                            transition modes, default, MaxSteps)">
    <name>
      <value>unnamed</value>
    </name>
  </net>
</pnml>
```

1. Markenmenge – eine Klasse, die von `Token` abgeleitet ist und eine Marke beschreibt.

2. Markierungsstruktur – eine Klasse, die von `MarkingStructure` abgeleitet ist und die Struktur von Marken beschreibt.

3. Kantentypen – eine Zeichenkette beschreibt, ob gewöhnliche Kanten oder Kanten nach der Klassifikation in Tab. 3.1 verwendet werden.

4. Schrittparameter – eine Klasse, die von `StepStructure` abgeleitet ist und die Zusammensetzung eines Schrittes beschreibt.

5. Abhängigkeitsparameter – eine Klasse, die von `StepVerifier` abgeleitet ist und einen Kandidaten für einen Schritt überprüft.

6. Auswahlparameter – eine Klasse, die von `StepSelector` abgeleitet ist und aus einer Menge von Schritten die aktivierten Schritte bestimmt.

Da der PNK Netze im Dateiformat PNML abspeichert, finden sich die konkreten Parameter des Netztyps auch in der PNML-Datei wieder. Beispielsweise beschreibt das Progr. 5.4 ein nach dem Petrinetz-Hyperwürfel parametrisiertes *leeres* Petrinetz. Der Petrinetz-Typ (PNML-Attribut `type`) *`"PNHCube()"`* erwartet die oben genannten sechs Parameter. Die ersten beiden Parameter bezeichenen hier schwarze Marken als Markenmenge und Multimenge als Markierungsstruktur. Der Petrinetz-Typ lässt erweiterte Kantentypen (Parameter 3) zu. Ein Schritt ist für diesen Petrinetz-Typ eine Multimenge von Schaltmodi (Param. 4). Außerdem sind alle Multimengen von Schaltmodi auch Schritte (Param. 5); aber nur die maximalen Schritte können geschaltet werden (Param. 6). Label werden wie sonst im PNK auch mit Hilfe der Netztyp-Deklaration für den speziellen Petrinetz-Typ *`"PNHCube"`* implementiert (vgl. Abschn. 5.3.3).

Zur Zeit sind für den PNK schwarze Marken und *coloured token* als Markenmengen implementiert; weiterhin sind Mengen, Multimengen und Sequenzen als Markierungsstruktur sowie gewöhnliche Kantentypen und Kantentypen nach Tab. 3.1 implementiert. Als Schaltregelparameter sind alle in den Abbildungen dieses Abschnittes aufgeführte Klassen konkreter Schaltregelparameter implementiert. Weitere Parameter des Petrinetz-Hyperwürfels lassen sich leicht implementieren, indem die bisher implementierten Parameter angepasst und erweitert werden.

Eine Implementierung der parametrisierten Schaltregel kann nicht laufzeiteffizient sein, da zu viele Abhängigkeiten zu beachten sind, die in konkreten Petrinetz-Typen im Allgemeinen nicht gemeinsam auftreten. Sie ist aber sehr wohl entwicklungszeiteffizient, da sie nur einmal implementiert, für jede Kombination von Parametern eines parametrisierten Petrinetz-Typs verwendet wird. Jeder Parameter dieses Petrinetz-Typs modifiziert die Schaltregel lediglich an bestimmten Stellen.

6 Zusammenfassung

In der vorgelegten Arbeit werden verschiedene Konzepte für einen allgemeinen Blick auf verschieden Petrinetz-Typen vorgestellt. Diese Konzepte erweisen ihren Nutzen beim Entwurf einer allgemeinen Petrinetz-Beschreibungssprache und einer allgemeinen Infrastruktur zum Bau von Petrinetz-Werkzeugen. Es wird herausgearbeitet, worin die Gemeinsamkeiten und Unterschiede verschiedener Petrinetz-Typen liegen.

Der Schwerpunkt der Arbeit liegt auf der software-technischen Unterstützung der zuvor präsentierten Konzepte. In einer ersten Sichtweise werden Petrinetze in Netzelemente (Stellen, Transitionen und Kanten) sowie Label getrennt. Es wird gezeigt, dass das daraus folgende Labelkonzept sich als Basis für eine Petrinetz-Beschreibungssprache und als Basis für eine Infrastruktur zum Bau von Petrinetz-Werkzeugen eignet.

Eine Petrinetz-Beschreibungssprache, wie die in dieser Arbeit vorgestellte *Petri Net Markup Language* (PNML), dient als allgemeines Austauschformat für Petrinetze. Die Akzeptanz einer Petrinetz-Beschreibungssprache wird erhöht einerseits durch vorhandene software-technische Unterstützung in Werkzeugen und andererseits durch einen Standard für verschiedene Label. Es bleibt Aufgabe einer Standardisierungskommission, einen in der Fachwelt akzeptierten Vorschlag dafür zu erarbeiten. Im Standard sollten die Bezeichner für Label und die Datentypen ihrer Werte festgelegt werden.

Eine Infrastruktur zum Bau von Petrinetz-Werkzeugen, wie der in dieser Arbeit vorgestellte Petrinetz-Kern (PNK), erleichtert den Bau von Petrinetz-Werkzeugen, indem Methoden und Werkzeuge zur Verwaltung von Netzelementen und Labeln implementiert werden. Durch eine gemeinsame Schnittstelle für alle Label kann der PNK Petrinetze jeden Typs bearbeiten und stellt einen grafischen Editor sowie je eine Lade- und Speicheroperation für das allgemeine Dateiformat PNML zur Verfügung. Mit weiteren Annahmen zu bestimmten Labeln gelingt sogar ein einfacher allgemeiner Markenspielsimulator.

Für weitergehende Unterstützung der verschiedenen Petrinetz-Typen in einer Infrastruktur zum Bau von Petrinetz-Werkzeugen benötigt man tiefergehende Konzepte. Der Petrinetz-Würfel war ein erster Ansatz eines parametrisierten Petrinetz-Typs, dessen Schaltregel in einem Markenspielsimulator implementiert wurde. Auf den Petrinetz-Würfel aufbauend wird in dieser Arbeit der Petrinetz-Hyperwürfel vorgestellt, im dem eine allgemeine Schaltregel und damit auch eine beliebige Schrittschaltregel für Petrinetze formuliert werden kann. Der Petrinetz-Hyperwürfel und seine Schaltregel wurden mit Hilfe des PNK implementiert. Damit wird gezeigt, dass es möglich

ist, ein allgemeines Petrinetz-Werkzeug mit Markenspielsimulator zu implementieren, ohne den konkreten Petrinetz-Typ zu kennen. Alle auf der Schaltregel von Petrinetzen basierenden Analysealgorithmen (z. B. Erreichbarkeitsanalyse) können so unabhängig von einem konkreten Petrinetz-Typ implementiert werden.

Der Petrinetz-Hyperwürfel umfasst ein wichtiges Teilkonzept zur Klassifikation von Kantentypen, das in dieser Arbeit vorgestellt wird. Dabei werden jeder Kante zwei Aspekte zugeordnet, die unabhängig voneinander betrachtet werden. Eine Kante eines Petrinetzes stellt einerseits eine Aktivierungsbedingung für ihre Transition dar. Andererseits bestimmt eine Kante den Effekt auf ihre Stelle, wenn ihre Transition schaltet.

Die hier vorgestellte Petrinetz-Beschreibungssprache PNML ist ein Beitrag zur Standardisierung von Petrinetzen. Sie enthält neben dem Labelkonzept weitere Konzepte. Das Seitenkonzept dient der Strukturierung und Separierung großer Petrinetze. Das Modulkonzept dient der Wiederverwendung gleichartiger Teilnetze. Beide Konzepte sind unabhängig von konkreten Petrinetz-Typen. Außerdem ist für PNML vorgesehen, Definitionen für Label als Bestandteile von Definitionen für Petrinetz-Typen aus einem allgemeinen Konventionen-Dokument zu entnehmen. Dies erleichtert den Austausch von Netzen ähnlicher Typen, da die Gemeinsamkeiten und Unterschiede zwischen den Typen damit herausgestellt werden.

Die Infrastruktur PNK ist eine Unterstützung für Petrinetz-Werkzeugentwickler. Mit Hilfe des PNK werden mit geringem Aufwand Petrinetz-Werkzeuge gebaut, die eine grafische Benutzeroberfläche, Lade- und Speicheroperationen (von bzw. nach PNML) sowie Methoden zur Verwaltung eines Petrinetzes enthalten. Ein Nutzer des PNK implementiert lediglich Label für bestimmte Petrinetz-Typen und Anwendungsmodule, die die Implementierungsaufgabe lösen.

Die vorgelegte Arbeit öffnet Raum für weitere anschließende Arbeiten. Die Petrinetz-Beschreibungssprache PNML greift auf ein Konventionen-Dokument zurück. Dieses Dokument enthält Definitionen von Labeln, die in verschiedenen Petrinetz-Typen verwendet werden. Ein zu erarbeitendes Konventionen-Dokument kann in Zukunft Petrinetz-Werkzeugentwicklern bei der Beschreibung von Petrinetz-Typen und beim Austausch mit anderen Petrinetz-Werkzeugen helfen. Es lohnt sich, ein solches allgemein akzeptiertes Konventionen-Dokument zu entwickeln.

Der PNK sollte sinnvollerweise durch eine Bibliothek ergänzt werden, die Labelimplementationen und Anwendungsmodule enthält. Damit wird es einem PNK-Werkzeugbauer einfach gemacht, Petrinetz-Typen und PNK-Werkzeuge aus vorhandenen Teilen zusammenzustellen. Wenn diese Bibliothek eine kritische Größe erreicht, kann es schwierig werden zu entscheiden, welches Anwendungsmodul für welchen Petrinetz-Typ zulässig ist. Abhilfe könnte ein Konzept schaffen, das die Anforderung eines Anwendungsmoduls an einen Petrinetz-Typ formulieren lässt. Beispielsweise erwartet ein Anwendungsmodul eine bestimmte Methode eines bestimmten Labels. Dann

kann das Anwendungsmodul für Netze jeden Petrinetz-Typs verwendet werden, der ein derartiges Label zulässt.

Idealerweise wird die Entwicklung eines solchen Konzeptes synchron zur Entwicklung eines Konventionen-Dokumentes für PNML verlaufen. In PNML standardisierte Label erhalten so eine Referenzimplementation im PNK.

Schließlich wird mit dem PNK-Werkzeug `PNHCube` ein universelles Petrinetz-Werkzeug für einen weitestmöglich parametrisierten Petrinetz-Typ vorgestellt. Ausgehend von dieser universellen Implementation lässt sich eine spezielle Implementation für einen konkreten Petrinetz-Typ ableiten. Ein solches PNK-Werkzeug kann genau zugeschnitten und effizient implementiert werden.

Gemeinsam bilden PNK und PNML ein mächtiges Grundgerüst zum Bau beliebiger Petrinetz-Werkzeuge. PNML leistet darüberhinaus einen Beitrag zur Kooperation verschiedener Petrinetz-Werkzeuge.

[154]

Literaturverzeichnis

Das Literaturverzeichnis ist nach DIN 1505 formatiert. Am Ende jeden Eintrags steht eine Liste von Seitenzahlen. Auf den entsprechenden Seiten der vorliegenden Arbeit wird auf diese Literatur verwiesen. Zu Literatur aus dem *World Wide Web* wird ein URL angegeben, unter dem das Dokument gefunden werden kann. Außerdem ist für diese Literaturstellen, sofern zu finden, das Datum der ersten Veröffentlichung (Monat und Jahr) und der letzten beobachteten Änderung angegeben.

[ADO00] VAN DER AALST, Wil (Hrsg.) ; DESEL, Jörg (Hrsg.) ; OBERWEIS, Andreas (Hrsg.): *Business Process Management: Models, Techniques, and Empirical Studies*. Berlin ; Heidelberg ; New York et al : Springer, 2000 (Lecture Notes in Computer Science (LNCS) 1806) (S. 2)

[AMBD98] AJMONE MARSAN, Marco ; BOBBIO, Andrea ; DONATELLI, Susanna: Petri Nets in Performance Analysis: An Introduction. In: [RR98], S. 211–256 (S. 25)

[AMC87] AJMONE MARSAN, Marco ; CHIOLA, Giovanni: On Petri Nets with Deterministic and Exponentially Distributed Firing Times. In: ROZENBERG, Grzegorz (Hrsg.): *Advances in Petri Nets 1987*. Berlin ; Heidelberg ; New York et al : Springer-Verlag, 1987 (Lecture Notes in Computer Science (LNCS) 266), S. 132–145 (S. 22)

[APN01] Lehrstuhl Informatik IV, Universität Dortmund: *APNN-Toolbox*. URL `http://www4.cs.uni-dortmund.de/APNN-TOOLBOX/`. August 2001. – 2001-08-28 (S. 121)

[ASN97] ISO/IEC International Standard 8824-1: *Information technology : Abstract Syntax Notation One (ASN.1) : Specification of basic notation*. URL `http://asn1.elibel.tm.fr/`. 1997. – ITU-T Recommendation X.680 (S. 84)

[Balo0] BALZERT, Helmut: *Lehrbuch der Software-Technik*. Bd. 1: *Software-Entwicklung*. 2. Aufl. Heidelberg ; Berlin : Spektrum, Akad. Verl., 2000. – ISBN 3–8274–0480–0 (S. 120)

[BBK+00] Bastide, Rémi (Hrsg.) ; Billington, Jonathan (Hrsg.) ; Kindler, Ekkart (Hrsg.) ; Kordon, Fabrice (Hrsg.) ; Mortensen, Kjeld H. (Hrsg.) ; 21st ICATPN (Veranst.): *Meeting on XML/SGML based Interchange Formats for Petri Nets.* Århus, Denmark, Juni 2000 (S. 29, 84, 161)

[BEH+01] Brandes, Ulrik ; Eiglsperger, Markus ; Herman, Ivan ; Himsolt, Michael ; Marshall, M. S.: GraphML Progress Report (Structural Layer Proposal). In: Mutzel, Petra (Hrsg.) ; Jünger, Michael (Hrsg.) ; Leipert, Sebastian (Hrsg.): *Graph Drawing.* Berlin ; Heidelberg ; New York et al : Springer, 2001 (Lecture Notes in Computer Science (LNCS) 2265), S. 501–512 (S. 113)

[Bes87] Best, Eike: Structure Theory of Petri Nets: the Free Choice Hiatus. In: [BRR87], S. 168–205 (S. 23)

[BF86] Best, Eike ; Fernández, César: Notations and Terminology on Petri Net Theory / Gesellschaft für Mathematik und Datenverarbeitung. 1986 (Arbeitspapiere der GMD 195). (S. 3, 25)

[Bil89] Billington, Jonathan: Many-sorted High-level Nets. In: *Proceedings of the 3rd International Workshop on Petri Nets and Performance Models*, IEEE Computer Society Press, Dezember 1989, S. 166–179 (S. 35)

[Bil91] Billington, Jonathan: *Extensions to Coloured Petri Nets and their Application to Protocols*, University of Cambridge, Technical Report No. 222, Mai 1991 (S. 46)

[Bil97] Billington, Jonathan: Development of an International Standard for High-level Petri Nets. In: *Proc. 3rd IEEE International Software Engineering Standards Symposium and Forum (ISESS'97).* Walnut Creek, 1997. – ISBN 0–8186–7837–2, S. 155–162 (S. 84)

[BKK95] Bause, Falko ; Kemper, Peter ; Kritzinger, Pieter: Abstract Petri Net Notation. In: *Petri Net Newsletter* (1995), Nr. 49, S. 9–27. – ISSN 0931–1084 (S. 29, 83, 84, 121)

[BM98] Behme, Henning ; Mintert, Stefan: *XML in der Praxis.* Bonn : Addison-Wesley-Longman, 1998. – ISBN 3–8273–1330–9 (S. 81, 101)

[BM00] Bruni, Roberto ; Montanari, Ugo: Executing Transactions in Zero-Safe Nets. In: Nielsen, Mogens (Hrsg.) ; Simpson, Dan (Hrsg.): *Application and Theory of Petri Nets 2000.* Berlin ; Heidelberg ; New York et al : Springer, 2000 (Lecture Notes in Computer Science (LNCS) 1825), S. 83–102 (S. 22, 23, 70)

[Bra80] Brauer, Wilfried (Hrsg.): *Net Theory and Applications*. Berlin ; Heidelberg ; New York et al : Springer-Verlag, 1980 (Lecture Notes in Computer Science (LNCS) 84) (S. 2, 25, 158, 159, 162, 164)

[Bra92] Braun, Petra: *Ein allgemeines Petrinetz*, Johann Wolfgang Goethe-Universität, Frankfurt, Diplomarbeit, April 1992 (S. 3, 26)

[BRR87] Brauer, Wilfried (Hrsg.) ; Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Petri Nets: Central Models and Their Properties*. Berlin ; Heidelberg ; New York et al : Springer-Verlag, 1987 (Lecture Notes in Computer Science (LNCS) 254) (S. 25, 156, 158, 159, 161, 162, 163, 164)

[BXM99] Martin, Bruce (Hrsg.) ; Jano, Bashar (Hrsg.) (World Wide Web Consortium – W3C): *WAP Binary XML Content Format*. URL `http://www.w3.org/TR/wbxml/`. Juni 1999. – 1999-06-24 (S. 84)

[Cla01] Clark, James. *TREX – Tree Regular Expressions for XML*. URL `http://www.thaiopensource.com/trex/`. 2001 (S. 99, 112)

[COS01] Ley GmbH: *COSA Workflow*. URL `http://www.cosa.de`. 2001. – ohne Datum (S. 2, 28)

[CPN00a] CPN group, University of Århus, Denmark: *Design/CPN*. URL `http://www.daimi.au.dk/designCPN/`. 2000. – 2001-09-21 (S. 2, 27, 28, 83, 88, 120)

[CPN00b] The MARS Team: *CPN-AMI*. URL `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`. 2000. – 2000-12-12 (S. 28, 121)

[DE95] Desel, Jörg ; Esparza, Javier: *Free Choice Petri Nets*. Cambridge University Press, 1995 (S. 23)

[DJ01] Desel, Jörg ; Juhás, Gabriel: "What is a Petri Net?" Informal Answers for the Informed Reader. In: [EJPR01], S. 1–25 (S. 121)

[DK99] Donatelli, Susanna (Hrsg.) ; Kleijn, Jetty (Hrsg.): *Application and Theory of Petri Nets 1999*. Berlin ; Heidelberg ; New York et al : Springer, 1999 (Lecture Notes in Computer Science (LNCS) 1639) (S. 159, 160)

[DKKO98] Desel, Jörg (Hrsg.) ; Kemper, Peter (Hrsg.) ; Kindler, Ekkart (Hrsg.) ; Oberweis, Andreas (Hrsg.): *5. Workshop Algorithmen und Werkzeuge für Petrinetze*. Universität Dortmund, Fachbereich Informatik, Oktober 1998 (Forschungsberichte 694). – ISSN 0933–6192 (S. 159, 160)

[DR98] Desel, Jörg ; Reisig, Wolfgang: Place/Transition Petri Nets. In: [RR98], S. 122–173 (S. 25, 60, 66, 70, 75)

[EJN97] Esser, Robert ; Janneck, Jörn W. ; Naedele, Martin: Using an Object-Oriented Petri Net Tool for Heterogeneous Systems Design: A Case Study. In: Desel, Jörg (Hrsg.) ; Kindler, Ekkart (Hrsg.) ; Oberweis, Andreas (Hrsg.): *4. Workshop Algorithmen und Werkzeuge für Petrinetze*, Humboldt-Universität zu Berlin, Oktober 1997 (Informatik-Berichte 85). – ISSN 0863–095, S. 1–6 (S. 21)

[EJPR01] Ehrig, Hartmut (Hrsg.) ; Juhás, Gabriel (Hrsg.) ; Padberg, Julia (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Unifying Petri Nets : Advances in Petri Nets*. Berlin ; Heidelberg ; New York et al : Springer, 2001 (Lecture Notes in Computer Science (LNCS) 2128) (S. 121, 157)

[EMC+99] Ehrig, Hartmut ; Mahr, Bernd ; Cornelius, Felix ; Grosse-Rhode, Martin ; Zeitz, Philip: *Mathematisch-strukturelle Grundlagen der Informatik*. Berlin ; Heidelberg ; New York et al : Springer, 1999 (Springer-Lehrbuch). – ISBN 3–540–63865–2 (S. 10, 11, 26, 36)

[Fel93] Feldbrugge, Frits: Petri net tool overview 1992. In: Rosenberg, Grzegorz (Hrsg.): *Proceedings Annual European Workshop: Applications and Theory of Petri Nets '91*. Berlin ; Heidelberg ; New York et al : Springer, 1993 (Lecture Notes in Computer Science (LNCS) 674) (S. 3, 25)

[FS98] Fowler, Martin ; Scott, Kendall: *UML – konzentriert : Die neue Standard-Objektmodellierungssprache anwenden*. Bonn : Addison-Wesley-Longman, 1998. – ISBN 3–8273–1329–5 (S. 117)

[Gar98] Garbe, Wolf: *Petri Net Tool Survey*. URL `http://home.arcor-online.de/wolf.garbe/petrisoft.html`. Juni 1998. – 1999-06-12 (S. 25)

[Gen87] Genrich, Hartmann J.: Predicate/Transition Nets. In: [BRR87], S. 207–247 (S. 15, 25, 37)

[GLT80] Genrich, Hartmann J. ; Lautenbach, Kurt ; Thiagarajan, Pazhamaneri S.: Elements of General Net Theory. In: [Bra80], S. 21–163 (S. 2, 25)

[Goo95] Goos, Gerhard: *Vorlesungen über Informatik*. Bd. 1: *Grundlagen und funktionales Programmieren*. Berlin ; Heidelberg ; New York et al : Springer, 1995. – (Springer-Lehrbuch). – ISBN 3–540–57281–3 (S. 7, 8)

[Gra97] Grahlmann, Bernd: The PEP Tool. In: *Tool Presentations of ATPN'97 (Application and Theory of Petri Nets)*, 1997. – `http://theoretica.informatik.uni-oldenburg.de/~pep/` (S. 2, 27)

[GS80] Genrich, Hartmann J. ; Stankiewicz-Wiechno, Ewa: A Dictionary of Some Basic Notions of Net Theory. In: [Bra80], S. 519–535 (S. 25)

[Haa98] Haar, Stefan: PEPINA: INA Net Analysis in the PEP Environment. In: [DKKO98], S. 50–54. – ISSN 0933–6192 (S. 27, 84)

[HL00] Hanisch, Hans-Michael ; Lüder, Arndt: A Signal Extension for Petri Nets and its Use in Controller Design. In: *Fundamenta Informaticae* 41 (2000), Nr. 4, S. 415–431 (S. 2, 17, 18, 46, 66, 88)

[HVA97] Hauschildt, Dirk ; Verbeek, Eric ; van der Aalst, Wil: WOFLAN: A Petri-net-based Workflow Analyzer / Eindhoven University of Technology. 1997 (Computing Science Reports 97-12). (S. 32)

[Jen83] Jensen, Kurt: High Level Petri Nets. In: Pagnoni, Anastasia (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Application and Theory of Nets*, Springer-Verlag, 1983 (Informatik-Fachberichte 66), S. 166–180 (S. 15)

[Jen87] Jensen, Kurt: Coloured Petri Nets. In: [BRR87], S. 248–299 (S. 25)

[Jen92] Jensen, Kurt: *Coloured Petri Nets*. Bd. 1: *Basic Concepts*. Berlin ; Heidelberg ; New York et al : Springer-Verlag, 1992. – ISBN 3–540–55597–8 (S. 2, 15, 16, 21, 32, 35, 36, 66, 74)

[JKW00] Jüngel, Matthias ; Kindler, Ekkart ; Weber, Michael: The Petri Net Markup Language. In: *Petri Net Newsletter* (2000), Nr. 59, S. 24–29. – ISSN 0931–1084 (S. 81)

[Juh99] Juhás, Gabriel: Reasoning about Algebraic Generalisation of Petri Nets. In: [DK99], S. 324–343 (S. 3, 26)

[KD96] Kindler, Ekkart ; Desel, Jörg: Der Traum von einem universellen Petrinetz-Werkzeug – Der Petrinetz-Kern. In: Desel, Jörg (Hrsg.) ; Oberweis, Andreas (Hrsg.) ; Kindler, Ekkart (Hrsg.): *3. Workshop Algorithmen und Werkzeuge für Petrinetze*, Institut AIFB, Universität Karlsruhe, Oktober 1996 (Forschungsberichte 341), S. 27–32 (S. 116)

[KG99] Katz, Shmuel ; Grumberg, Orna: VeriTech: Translating among Specifications and Verification Tools / The Technion, Haifa, Israel. 1999. – Technical Report (S. 28, 88, 120)

[Küh99] Kühnel, Ralf: *Die Java-2-Fibel : Programmierung von Threads und Applets*. 3., akt. und erw. Aufl. Bonn : Addison Wesley Longman, 1999. – ISBN 3–8273–1410–0 (S. 119)

[Kin97] Kindler, Ekkart: Der Petrinetz-Kern: Ein Traum wird wahr. In: Ehrig, H. (Hrsg.) ; Reisig, W. (Hrsg.) ; Weber, H. (Hrsg.): *Move-On-Workshop der DFG-Forschergruppe Petrinetz-Technologie*, Technische Universiät Berlin, 1997, S. 121–124 (S. 116)

[KMW99] Kummer, Olaf ; Moldt, Daniel ; Wienberg, Frank: Symmetric Communication between Coloured Petri Net Simulations and Java-Processes. In: [DK99], S. 86–105 (S. 21)

[KO98] Kindler, Ekkart ; Oschmann, Frank: The Petri Net Kernel: An INA-Pilot. In: [DKKO98], S. 62–66. – ISSN 0933–6192 (S. 27, 84)

[Krä85] Krämer, Bernd: Stepwise Construction of Non-Sequential Software Systems Using a Net-Based Specification Language. In: Rozenberg, Grzegorz (Hrsg.): *Advances in Petri Nets 1984*. Berlin ; Heidelberg ; New York et al : Springer-Verlag, 1985 (Lecture Notes in Computer Science (LNCS) 188), S. 307–330 (S. 35)

[KSW97] Kindler, Ekkart ; Schwenzer, Ines ; Weber, Michael. *Der Petrinetz-Kern: Dokumentation der Anwendungs-Schnittstelle (V. 0.8)*. 1997 (S. 116)

[KW98] Kindler, Ekkart ; Weber, Michael: The Dimensions of Petri Nets: The Petri Net Cube. In: *Bulletin of EATCS* (1998), Oktober, Nr. 66, S. 155–166 (S. 3, 26, 31, 34, 35, 36, 40, 41)

[KW99] Kindler, Ekkart ; Weber, Michael: *The Petri Net Kernel : Documentation of the Application Interface : PNK Version 2.0*. Humboldt-Universität zu Berlin, Institut für Informatik, Januar 1999. – `http://www.informatik.hu-berlin.de/top/pnk/` (S. 117, 137)

[KW01] Kindler, Ekkart ; Weber, Michael: The Petri Net Kernel – An Infrastructure for Building Petri Net Tools. In: *Software Tools for Technology Transfer (STTT)* 3 (2001), September, Nr. 4, S. 486–497. – ISSN 1433–2779 (S. 116, 117)

[KWD01] Kummer, Olaf ; Wienberg, Frank ; Duvigneau, Michael: *Renew – User Guide*. URL `http://www.informatik.uni-hamburg.de/TGI/renew/renew.pdf`. Juli 2001. – Release 1.5.2 (S. 21)

[Lau87] Lautenbach, Kurt: Linear Algebraic Techniques for Place/Transition Nets. In: [BRR87], S. 142–167 (S. 14)

[LC94] Lakos, Charles ; Christensen, Søren: A General Systematic Approach to Arc Extensions for Coloured Petri Nets. In: Valette, Robert (Hrsg.): *Application an Theory of Petri Nets 1994*. Berlin ; Heidelberg ; New York et al : Springer-Verlag, 1994 (Lecture Notes in Computer Science (LNCS) 815), S. 338–357 (S. 3, 25, 47)

[LF97] von Löwis, Martin ; Fischbeck, Nils: *Das Python-Buch : Referenz der objektorientierten Skriptsprache für GUIs und Netzwerke*. Bonn : Addison-Wesley-Longman, 1997. – ISBN 3–8273–1110–1 (S. 116)

[LM98] Lyngsø, Regnar B. ; Mailund, Thomas: Textual Interchange Format for High-Level Petri Nets. In: Jensen, K. (Hrsg.): *Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, Århus University, Denmark, Juni 1998 (Daimi PB 532), S. 47–64 (S. 29)

[LO01] Lenz, Kirsten ; Oberweis, Andreas: Modeling Interorganizational Workflows with XML Nets. In: *Proceedings of the Hawai'i International Conference On System Sciences*. Maui, Hawaii, Januar 2001 (S. 26)

[Mar98] Theoretical Computer Science Laboratory, Helsinki University of Technology: *The Maria Project*. URL `http://saturn.hut.fi/html/Maria.html`. 1998. – 2001-01-07 (S. 27, 121)

[MCK99] *Model-Checking Kit*. URL `http://wwwbrauer.in.tum.de/gruppen/theorie/KIT/`. August 1999. – 2002-01-11 (S. 28, 120)

[McM93] McMillan, Kenneth L.: *Symbolic Model Checking*. Kluwer Academic Publishers, 1993 (S. 89)

[MM90] Meseguer, José ; Montanari, Ugo: Petri Nets are Monoids. In: *Information and Computation* (1990), Nr. 88, S. 105–155 (S. 26)

[MM00] Mailund, Thomas ; Mortensen, Kjeld H.: Separation of Style and Content with XML in an Interchange Format for High-level Petri Nets. In: [BBK+00], S. 7–11 (S. 86)

[Pad96] Padberg, Julia: *Abstract Petri Nets : Uniform Approach and Rule-Based Refinement*, Technische Universität Berlin, Diss., Februar 1996 (S. 3, 26)

[Pag87] Pagnoni, Anastasia: Stochastic Nets and Performance Evaluation. In: [BRR87], S. 460–478 (S. 25)

[PEP98] Stehno, Christian (Hrsg.): *PEP Homepage*. URL `http://theoretica.informatik.uni-oldenburg.de/~pep/`. 1998. – 2001-12-13 (S. 28, 83, 120)

[Pet77a] Peterson, James L.: Petri Nets. In: *Computing Surveys* 9 (1977), September, Nr. 3, S. 223–252 (S. 46, 49)

[Pet77b] Petri, Carl Adam: General Net Theory. In: Shaw, B. (Hrsg.): *Computing System Design: Proc. of the Joint IBM University of Newcastle upon Tyne Seminar, Sep., 1976*, University of Newcastle upon Tyne, 1977, S. 131–169 (S. 2, 25)

[Pet80] Petri, Carl Adam: Introduction to General Net Theory. In: [Bra80], S. 1–19 (S. 25)

[PNK00] Fischer, Erik (Hrsg.): *Petri Net Kernel*. URL `http://www.informatik.hu-berlin.de/top/pnk/`. 2000. – 2002-05-06 (S. 32, 88, 116, 129, 134)

[PNM00] Weber, Michael (Hrsg.) ; Jüngel, Matthias (Hrsg.): *Petri Net Markup Language*. URL `http://www.informatik.hu-berlin.de/top/pnml/`. 2000. – 2001-07-19 (S. 81, 99, 112)

[PNT00] Mortensen, Kjeld H. (Hrsg.): *Petri Nets Tool Database*. URL `http://www.daimi.aau.dk/PetriNets/tools/db.html`. 2000. – 2002-05-15 (S. 3, 25)

[PS92] Posthoff, Christian ; Schulz, Konrad: *Grundkurs Theoretische Informatik*. Stuttgart ; Leipzig : B. G. Teubner Verlagsgesellschaft, 1992. – ISBN 3–8154–2036–9 (S. 7)

[Ram74] Ramchandani, Chander: *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*, Cambridge, Mass.: MIT, Dept. Electrical Engineering, Ph. D. Thesis, 1974 (S. 21, 75)

[RE98] Rozenberg, Grzegorz ; Engelfriet, Joost: Elementary Net Systems. In: [RR98], S. 12–121 (S. 25)

[Rei86] Reisig, Wolfgang: *Petrinetze : Eine Einführung*. 2. Aufl. Berlin ; Heidelberg ; New York et al : Springer, 1986 (Studienreihe Informatik) (S. 34)

[Rei87] Reisig, Wolfgang: Place/Transition Systems. In: [BRR87], S. 117–141 (S. 2, 14, 25, 36)

[Rei91a] Reisig, Wolfgang: Concurrent temporal logic / Technische Universität München. 1991 (SFB-Bericht 342/7/91B). (S. 2)

[Rei91b] Reisig, Wolfgang: Petri Nets and Algebraic Specifications. In: *Theoretical Computer Science* (1991), Mai, Nr. 80, S. 1–34 (S. 17, 35)

[Rei98] Reisig, Wolfgang: *Elements of Distributed Algorithms : Modeling and Analysis with Petri Nets*. Berlin ; Heidelberg ; New York et al : Springer, 1998 (S. 17, 70)

[REL01] Clark, James (Hrsg.) ; Murata, Makoto (Hrsg.) (Organization for the Advancement of Structured Information Standards – OASIS): *RELAX NG*. URL `http://www.oasis-open.org/committees/relax-ng/`. Dezember 2001. – 2002-04-03 (S. 99)

[Ren99] Theoretische Grundlagen der Informatik, Universität Hamburg: *Renew : The Reference Net Workshop*. URL `http://www.renew.de/`. März 1999. – 2001/07/03 (S. 21, 27, 29)

[Roc01] Roch, Stephan: Reducing Simultaenous Firing in Signal-Net Systems with Strongly Connected Sets. In: Czaja, Ludwik (Hrsg.): *Workshop Concurrency, Specification & Programming : Proceedings* Warsaw University, 2001, S. 230–241 (S. 17, 18, 70)

[Rou87] Roucairol, Gérard: FIFO-Nets. In: [BRR87], S. 436–459 (S. 25, 37)

[Roz87] Rozenberg, Grzegorz: Behaviour of Elementary Net Systems. In: [BRR87], S. 60–94 (S. 15, 25)

[Roz92] Rozenberg, Grzegorz (Hrsg.): *Advances in Petri Nets 1992*. Berlin ; Heidelberg ; New York et al : Springer-Verlag, 1992 (Lecture Notes in Computer Science (LNCS) 609) (S. 2)

[RR98] Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Lectures on Petri Nets I: Basic Models*. Berlin ; Heidelberg ; New York et al : Springer, 1998 (Lecture Notes in Computer Science (LNCS) 1491) (S. 25, 155, 158, 162, 164)

[RS98] Roch, Stephan ; Starke, Peter: INA : Integrierter Netz Analysator Version 2.1 / Humboldt Universität zu Berlin. 1998 (Informatik-Berichte 101). . (siehe auch URL `http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina/`). – ISSN 0863–095 (S. 27, 83, 87)

[Ruc96] Ruckdeschel, Wilhelm: *Modellierung regelbasierten Pilotenverhaltens mit Petrinetzen*, Universität der Bundeswehr München, Diss., 1996 (S. 87)

[Sch99] Schulz, Reiner. *The Petri Net Kernel : File Format Documentation*. URL `http://www.informatik.hu-berlin.de/top/pnk/literatur/pnkFileFormatExt.ps`. März 1999 (S. 137)

[Sch00] Schmidt, Karsten. *LoLA – a Low Level Petri net Analyzer*. URL `http://www.informatik.hu-berlin.de/~kschmidt/lola.html`. September 2000 (S. 27, 28, 83, 87)

[Sif80] Sifakis, Joseph: Performance Evaluation of Systems Using Nets. In: [Bra80], S. 307–319 (S. 25)

[SM83] Suzuki, Ichiro ; Murata, Tadao: A Method for Stepwise Refinement and Abstraction of Petri Nets. In: *Journal of Computer and System Science* 27 (1983), Nr. 1, S. 51–76 (S. 87)

[SMB97] Steffen, Bernhard ; Margaria, Tiziana ; Braun, Volker: The Electronic Tool Integration Platform : Concepts and Design. In: *International Journal on Software Tools for Technology Transfer (STTT)* 1 (1997), Nr. 1+2, S. 9–30. – `http://www.eti-service.org/` (S. 28, 120)

[Smi98] Smith, Einar: Principles of High-Level Net Theory. In: [RR98], S. 174–210 (S. 25)

[Sta90] Starke, Peter H.: *Analyse von Petri-Netz-Modellen*. Stuttgart : B. G. Teubner, 1990 (Leitfäden und Monographien der Informatik) (S. 15, 24, 26, 66)

[Sta95] Starke, Peter H.: A Memo on Time Constraints in Petri Nets / Humboldt-Universiät zu Berlin, Institut für Informatik. 1995 (Informatik-Berichte 46). . – ISSN 0863–095 (S. 2, 3, 21, 25, 58, 69, 70)

[Stö97] Störrle, Harald: An Evaluation of High-End Tools for Petri-Nets / Ludwig-Maximilians-Universität München. 1997 (Technischer Bericht 9802). (S. 3, 25)

[SVG01] Lilley, Chris (Hrsg.) ; Jackson, Dean (Hrsg.) (World Wide Web Consortium – W3C): *Scalable Vector Graphics (SVG)*. URL `http://www.w3.org/Graphics/SVG`. 2001. – 2002-05-01 (S. 113)

[Thi87] Thiagarajan, Pazhamaneri S.: Elementary Net Systems. In: [BRR87], S. 26–59 (S. 15, 25, 37)

[Val79] Valette, Robert: Analysis of Petri Nets by Stepwise Refinements. In: *Journal of Computer and System Sciences* (1979), Nr. 18, S. 35–46 (S. 87)

[Val98] Valk, Rüdiger: Petri Nets as Token Objects – An Introduction to Elementary Object Nets. In: Desel, Jörg (Hrsg.) ; Silva, Manuel (Hrsg.): *Application and Theory of Petri Nets 1998*. Berlin ; Heidelberg ; New York et al : Springer, 1998 (Lecture Notes in Computer Science (LNCS) 1420), S. 1–25 (S. 19, 20, 78)

[Vog93] Vogler, Walter: Bisimulation and action refinement. In: *Theoretical Computer Science* (1993), Nr. 114, S. 173–200 (S. 87)

[VSY98] Vogler, Walter ; Semenov, Alexei ; Yakovlev, Alexander V.: Unfolding and Finite Prefix for Nets with Read Arcs. In: Sangiorgi, Davide (Hrsg.) ; de Simone, Robert (Hrsg.): *CONCUR'98 Concurrency Theory*. Berlin ; Heidelberg ; New York et al : Springer, 1998 (Lecture Notes in Computer Science (LNCS) 1466), S. 501–516 (S. 46)

[Web97] Weber, Michael: CASSY: Einige Gründe zur Verwendung von Petrinetzen. In: Ehrig, Hartmut (Hrsg.) ; Reisig, Wolfgang (Hrsg.) ; Weber, Herbert (Hrsg.): *Move-On-Workshop der DFG-Forschergruppe Petrinetz-Technologie*, Technische Universität Berlin, 1997 (Forschungsberichte des Fachbereiches Informatik 97-21), S. 167–173 (S. 87)

[Web99] Weber, Michael: Der Petrinetz-Würfel im Petrinetz-Kern. In: Desel, Jörg (Hrsg.) ; Oberweis, Andreas (Hrsg.): *6. Workshop Algorithmen und Werkzeuge für Petrinetze*, J. W. Goethe-Universität Frankfurt/Main, Institut für Wirtschaftsinformatik, Oktober 1999, S. 69–74 (S. 26, 140)

[Wer00] Wernicke, Mike: *Integration von PEP-Dateien in den PNK*, Humboldt-Universität zu Berlin, Studienarbeit, 2000 (S. 84)

[WWV+97] Weber, Michael ; Walter, Rolf ; Völzer, Hagen ; Vesper, Tobias ; Reisig, Wolfgang ; Peuker, Sibylle ; Kindler, Ekkart ; Freiheit, Jörn ; Desel, Jörg: DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen / Humboldt-Universität zu Berlin. 1997 (Informatik-Berichte 88). . – ISSN 0863–095 (S. 17)

[XML00] Sperberg-McQueen, Michael (Hrsg.) ; Thompson, Henry (Hrsg.) (World Wide Web Consortium – W3C): *XML Schema*. URL `http://www.w3.org/XML/Schema`. April 2000. – 2002-03-22 (S. 98, 105)

Stichwortverzeichnis

Zu Beginn werden wichtige Symbole mit einem Verweis auf das korrespondierende Stichwort aufgeführt. Jedes Stichwort verweist auf Seiten der Arbeit, auf denen der Begriff vorzugsweise behandelt wird. Meist ist nur der Beginn der entsprechenden Abhandlung angegeben. Eine *kursiv* gesetzte Seitenzahl verweist auf die formale Definition des entsprechenden Stichwortes.

R

S

172

Erklärung

Hiermit erkläre ich, die vorliegende Dissertation „Allgemeine Konzepte zur softwaretechnischen Unterstützung verschiedener Petrinetz-Typen“ selbständig und ohne fremde Hilfe verfasst zu haben sowie nur die angegebene Literatur und die angegebenen Hilfsmittel verwendet zu haben.

Berlin, den 3. Juni 2002

[174]

Lebenslauf

Persönliche Daten

Michael Weber
Holteistr. 32a
10 245 Berlin

Tel.: (030) 2 92 03 27
E-Mail: MJWeber@gmx.net

Geb. am 15. Mai 1968 in Mühlhausen/Thür., deutsch
Ledig (in Lebensgemeinschaft), 2 Söhne (geb. 1997, 2002)

Schulbildung

09/1974–07/1986 Polytechnische Oberschule (Grundschule) und Erweiterte Oberschule (Gymnasium) in Mühlhausen/Thür.; erfolgreiche Teilnahme an Mathematikolympiaden im Kreis und Bezirk

Wehrdienst

10/1986–08/1989 Berlin

Studium

09/1989–07/1992 Grundstudium Informatik an der Technischen Universität Karl-Marx-Stadt/Chemnitz; Vordiplom

10/1992–01/1997 Hauptstudium Informatik an der Humboldt-Universität zu Berlin; Diplomarbeit: »*unless*-Aspekte und ihr Beitrag zur Verifikation von mit Petrinetzen modellierten Systemen«

Wissenschaftlicher Werdegang

05/1993–02/1994 Studentische Hilfskraft am Lehrstuhl Automaten und Systemtheorie (Prof. Starke), Humboldt-Universität zu Berlin

10/1995–09/1996 Leitung des Projekttutoriums »Informatik – Maschinisierung von Kopfarbeit«, Humboldt-Universität zu Berlin

02/1997-03/2002	Wissenschaftlicher Mitarbeiter im Projekt »Forschergruppe Petrinetz-Technologie«, gefördert durch die Deutsche Forschungsgemeinschaft (DFG), am Lehrstuhl Theorie der Programmierung (Prof. Reisig), Humboldt-Universität zu Berlin; Durchführung von Lehrveranstaltungen zur Geschäftsprozessmodellierung, zur Praktischen Informatik (mit Java) und zu LaTeX

Ausgeübte Tätigkeiten und Praktikum

02/1995–04/1995	Praktikum im Ingenieurbüro für Kunst und Technik II, Berlin; Aufgaben: Häufigkeitsbasierte Studie zu Tastatureingaben körperlich behinderter Menschen, Implementierung eines Prototypen
seit 02/1997	Gutachter für wissenschaftliche Konferenzen und Zeitschriften

Stipendium

04/1991–09/1996	Stipendiat eines Begabtenförderungswerkes

Auslandsaufenthalt

02/1994–06/1994	Intensivkurs Tschechisch an der Karls-Universität Prag

Fremdsprachen

Englisch, Tschechisch, Russisch

Interessen

Literatur, Fechten

Sonstiges

seit 01/1993	Mitglied der Gesellschaft für Informatik (GI)
seit 02/2002	Präsident der Fechtgemeinschaft Rotation Berlin

Berlin, im Januar 2003